Science of Computer Programming 155 (2018) 52–75

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Typechecking protocols with Mungo and StMungo: A session type toolchain for Java

Dimitrios Kouzapas [a], Ornela Dardha [b], Roly Perera [b,c], Simon J. Gay [b]

[a] Department of Computer Science, University of Cyprus, Cyprus
[b] School of Computing Science, University of Glasgow, UK
[c] School of Informatics, University of Edinburgh, UK

ARTICLE INFO

Article history:
Received 4 January 2017
Received in revised form 13 October 2017
Accepted 16 October 2017
Available online 5 December 2017

Keywords:
Session types
Object-oriented programming
Typestate
Type inference

ABSTRACT

Static typechecking is an important feature of many standard programming languages. However, static typing focuses on data rather than communication, and therefore does not help programmers correctly implement communication protocols in distributed systems. The theory of session types provides a basis for tackling this problem; we use it to develop two tools that support static typechecking of communication protocols in Java. The first tool, Mungo, extends Java with *typestate* definitions, which allow classes to be associated with state machines defining permitted sequences of method calls: for example, communication methods. The second tool, StMungo, takes a session type describing a communication protocol, and generates a typestate specification of the permitted sequences of messages in the protocol. Protocol implementations can be validated by Mungo against their typestate definitions and then compiled with a standard Java compiler. The result is a toolchain for static typechecking of communication protocols in Java. We formalise and prove soundness of the typestate inference system used by Mungo, and show that our toolchain can be used to typecheck a client for the standard Simple Mail Transfer Protocol (SMTP).

1. Introduction

Many popular programming languages use static typechecking to ensure correct manipulation of data. This offers significant practical benefit to software developers. However, modern software is increasingly reliant on communication, which must also be programmed correctly: messages must be sent and received in the correct sequence and with the correct format, in order to implement a desired communication protocol. The theory of session types [25,47] provides a basis for supporting communication-based programming by static typechecking of communication operations; it allows the structure of communication to be codified as type definitions, analogous to data type definitions, which can be used by compilers for typechecking. However, session types are not yet a feature of standard mainstream programming languages, so software developers are not able to benefit from them.

We present two tools that use the theory of session types to support static typechecking of communication protocols in Java. Our first tool, Mungo,[1] extends Java with *typestate* definitions, which associate classes with state machines defin-

E-mail address: Dimitrios.Kouzapas@glasgow.ac.uk (D. Kouzapas).

[1] Saint Mungo is the founder and patron saint of the city of Glasgow.

https://doi.org/10.1016/j.scico.2017.10.006

ing permitted sequences of method calls [45]. To associate a typestate definition with a class, the programmer adds a `@Typestate` annotation to the class telling Mungo where to find the typestate definition file. Mungo then ensures that instances of the class are used in a manner consistent with the declared typestate. To control aliasing, objects with typestate definitions are used according to a linear type discipline; this is analogous to the linear control of channels in session type systems. Our second tool, StMungo (Scribble-to-Mungo), uses this typestate feature to connect Java to the broader setting of communication protocols specified as session types in the Scribble protocol language [43]. Given a Scribble protocol projected to a particular endpoint (a so-called *local protocol*), StMungo generates a typestate specification capturing the sequences of sends and receives permitted along that endpoint. Each endpoint implementation can be validated separately by Mungo against its typestate definition and then compiled as usual with `javac`. The separate typechecking of each endpoint is integral to our approach, and is justified by the theory of *multiparty session types* [26], the formal foundation of Scribble. Multiparty session types provide an important safety guarantee: once each endpoint implementation is known to conform to its local protocol, the various implementations can be composed into a system free of communication errors.

Our work advances a line of research applying session types to real-world programming languages [9,15,17,16,22,29,34,35,37,41]. In particular, we build on the work of Gay et al. [23], which connected session types to the object-oriented notion of typestate. They observed that the valid sequences of messages for a given endpoint could be captured by a typestate definition for a class, allowing the channel endpoint to be modelled as an object. While an important idea, this earlier work lacked a practical implementation and relied on typestate declarations on parameters and return types. The Mungo/StMungo toolchain is the first integration of session types and a practical typestate system for checking the communication behaviour of Java programs. Moreover, Mungo uses a typestate inference system to eliminate the need for typestate declarations on parameters and return types. The Mungo/StMungo toolchain offers other practical advances over previous efforts to combine session types with objects. For example, SJ [29] only supports binary session types, whereas StMungo generates Mungo specifications from multiparty session types. Furthermore, Mungo permits objects with typestates to be stored in fields, whereas SJ requires them to be created and fully used within methods. Using the `@Typestate` annotation means we avoid any need for language extensions.

1.1. Contributions

Mungo. We describe the Mungo typestate checker for Java (§3). Mungo currently supports a subset of Java; support for the full language is discussed in §8.
StMungo. We describe StMungo (§2), which translates Scribble local protocols into Mungo typestate specifications. StMungo also generates Java method stubs for each endpoint. Both tools can be downloaded from [1].
SMTP case study. We present a statically typechecked SMTP client (§4), which illustrates the toolchain provided by Scribble, StMungo and Mungo.
Typestate inference system. We formalise the essential features of Mungo as a core object-oriented calculus (§5). We define a typestate inference system for that language and prove its type safety (§6).

A summary of this work was presented at PPDP 2016 [31].

2. StMungo: Scribble-to-Mungo

The integration of session types and typestate, defined by Gay et al. [23], consists of a formal translation of session types for communication channels into typestate specifications for channel objects. The main idea is that a channel object has methods for sending and receiving messages and the typestate specification defines the order in which these methods can be called; therefore it is a specification of the permitted sequences of messages, i.e. a channel protocol.

We extend this translation from binary to multiparty session types [26] and implement it as the StMungo (Scribble to Mungo) tool, which translates Scribble [43,49] local protocols into typestate specifications and skeleton socket-based implementation code. The resulting code is typechecked using Mungo. A Scribble local protocol describes the communication between one role and all the other participants in a multiparty scenario, including the way in which messages sent to different participants are interleaved. This interleaving is not captured by binary session types and by tools based on them, like SJ [29]. StMungo is based on the principle that each role in the multiparty communication can be abstracted as a Java class following the typestate corresponding to the role's local protocol. The typestate specification generated from StMungo together with the Mungo typechecker can guide the user in the design and implementation of distributed multiparty communication-based programs with guarantees on communication safety and soundness. StMungo is the first tool to provide a practical embedding of Scribble multiparty protocols into object-oriented languages with typestate.

We illustrate StMungo on a multiparty protocol that models the process of booking flights through a university travel agent. There are three participants: Researcher (abbreviated `R`), who intends to travel; Agent (`A`), who is able to make travel reservations; and Finance (`F`), who approves expenditure from the budget. After the `request`, `quote` and `check` messages requesting authorisation for a trip, Finance can choose to `approve` or `refuse` the request. The *global* protocol is defined as follows.

```
global protocol BuyTicket(role R, role A, role F){
  request(Travel) from R to A; quote(Price) from A to R;
  check(Price) from R to F;
  choice at F {
    approve(Code) from F to R,A;
    ticket(String) from A to R;
    invoice(Code) from A to F;
    payment(Price) from F to A; }
  or { refuse(String) from F to R,A; } }
```

The Scribble tool is used to check the above protocol definition for well-formedness and to derive a local version of the protocol for each role, according to the multiparty session types theory [26]. This is known as *endpoint projection*. Here we show the local protocol for Researcher, which describes only the messages involving that role. The `self` keyword indicates that `R` is the local endpoint.

```
local protocol BuyTicket_R(self R,role A,role F){
  request(Travel) to A;
  quote(Price) from A;
  check(Price) to F;
  choice at F {
    approve(Code) from F;
    ticket(String) from A; }
  or {
    refuse(String) from F; } }
```

Notice that the exchange of `invoice` and `payment` between Agent and Finance is not included. Similarly, the local projection for Agent omits the `check` message and the one for Finance omits the `request`, `quote` and `ticket` messages. StMungo converts the `BuyTicket_R` local protocol into the `RProtocol` typestate protocol:

```
typestate RProtocol {
State0 = { void send_requestTravelToA(Travel): State1 }
State1 = { Price receive_quotePriceFromA(): State2 }
State2 = { void send_checkPriceToF(Price): State3 }
State3 = { Choice1 receive_Choice1LabelFromF():
                              <APPROVE:State4, REFUSE:State6> }
State4 = { Code receive_approveCodeFromF(): State5 }
State5 = { String receive_ticketStringFromA(): end }
State6 = { String receive_refuseTravelFromF(): end } }
```

StMungo generates an API for this role, class `RRole`, which provides an implementation of `RProtocol`. When instantiated, it connects to the other role objects in the session (`ARole` and `FRole`). The method calls, describing the messages exchanged with the other roles, follow the interleaving specified by the `RProtocol` typestate. Alternatively, the developer may choose to ignore this API (and the Mungo socket library that it depends on), and use only the generated typestate protocols to develop his/her own implementation. He/she also has the ability to further refine the generated state machine, e.g., give appropriate names to states, or use anonymous states to have a coarser state refinement.

3. Mungo

Mungo extends Java with an optional *typestate* definition. The tool is implemented in Java using the JastAdd framework [24], a meta-compiler based on reference attribute grammars. Source files are typechecked in two phases: first according to the regular Java type system, and then according to our typestate extension. The source files can then be compiled using `javac` and executed in the standard Java 1.8 runtime environment.

A typestate is attached to a Java class and it defines an object protocol in the form of a state machine. Each state offers a set of methods that must be a subset of the methods defined by the class; each method specifies a transition to a successor state. Typestate definitions are provided in separate files, using the Java-like syntax, shown in Example 1 below. A typestate definition is attached to a class using the annotation `@Typestate("ProtocolName")`, where `"ProtocolName"` names the file where the typestate is defined. The typestate inference algorithm, presented in §6, constructs the sequences of methods called on all objects associated with a typestate, and then checks if the inferred typestate is a subtype of the object's declared typestate. Some Java features are not yet supported. Some we anticipate to be relatively straightforward extensions (`synchronised` statements, the conditional operator `?:`, inner and anonymous classes, and static initialisers). Generics, inheritance and exceptions are non-trivial and are discussed in future work (§8). Currently, generics are not supported; inheritance is supported for classes without typestate definitions; and exceptions are supported syntactically but are typechecked under the (unsound) assumption that no exceptions are thrown. (A `try-catch` statement is typechecked by typechecking the `try` body; if an exception is thrown a typestate violation may result.)

Example 1. We introduce Mungo through the example of an unbounded stack data structure that follows a typestate specification. Given the following enumerated type:

```
enum Check { EMPTY, NONEMPTY }
```

then one possible typestate protocol for a stack is as follows:

```
typestate StackProtocol {
  Empty    = {void push(int): NonEmpty,
              void deallocate(): end}
  NonEmpty = {void push(int): NonEmpty,
              int pop(): Unknown}
  Unknown  = {void push(int): NonEmpty,
              Check isEmpty(): <EMPTY: Empty,
                                NONEMPTY: NonEmpty>}}
```

This definition specifies that a stack is initially `Empty`. The `Empty` state declares two methods: `push(int)` pushes an integer onto the stack and proceeds to the `NonEmpty` state; `deallocate()` frees any resources used by the stack and terminates its usage. The `deallocate()` method is not available in any other state, requiring a client to empty the stack before it is done using it. In the `NonEmpty` state a client can either `push()` an element onto the stack and remain in the same state, or `pop()` an element from the stack and transition to `Unknown`. Unlike `push()`, `pop()` must leave the stack in the `Unknown` state because the number of elements on the stack is not tracked by the protocol. From the `Unknown` state, one can either `push()` and proceed to `NonEmpty`, or call `isEmpty()` to explicitly test whether the stack is empty. Calling `isEmpty()` returns a member of the enumeration `Check` defined earlier. This idiom, based on Java enumerations, is the mechanism for communicating a choice made by the callee synchronously back to the client, and is explained in more detail below. Here, a stack implementation can choose between returning `EMPTY` and transitioning to `Empty`, or returning `NONEMPTY` and transitioning to `NonEmpty`. We can now define a stack implementation `Stack` that conforms to the `StackProtocol` specification, using an integer array to store the elements. The `@Typestate("StackProtocol")` annotation is used to associate the typestate definition with the class:

```
@Typestate("StackProtocol")
class Stack {
  private int[] stack; private int head;
  Stack() { stack = new int[MAX]; head = 0; }
  void push(int d) { stack[head++] = d; }
  int pop() { return stack[head--]; }
  Check isEmpty() {
    if(head == 0) return Check.EMPTY;
    return Check.NONEMPTY);
  void deallocate() {} }
```

Finally, having implemented `StackProtocol`, we can define a stack client that uses the `Stack` with Mungo verifying that `Stack` instances are used correctly.

```
class StackUser {
  Stack pushN(Stack s, int n) {
    do { s.push(n--); }
    while(n>0);
    return s; }
  Stack popAll(Stack s)
  { loop : do {
      System.out.println(s.pop());
      switch(s.isEmpty()) {
        case EMPTY: break loop;
        case NONEMPTY: continue loop;
    } while(true);
    return s; }
  public static void main(String[] args)
  { StackUser su = new StackUser();
    Stack s = new Stack();
    Stack s2;
    s = su.pushN(s,16);
    s2 = su.popAll(s);
    s = su.pushN(s2,64);
```

```
21      s = su.popAll(s);
22      s.deallocate(); } }
```

For illustrative purposes, the client defines two helper methods. The first method, `pushN(Stack s, int n)`, for any $n > 0$, pushes the integers $n, \ldots, 1$ onto the stack `s`. The second, `popAll(Stack s)`, pops all the elements of `s`. We now discuss some details of the programming model, drawing on this example where appropriate.

Local variables, parameters, and return values. The `main()` method above creates a single `Stack` instance, stores it in a local variable `s`, and then passes it to various invocations of `pushN` and `popAll`, from which it is also returned as a result. We also make use of the additional local variable `s2`. When returned from a method, the stack has a potentially different typestate than it did as an argument. No explicit typestate definitions are required for the parameter or return types of `pushN` and `popAll`, since Mungo can infer them. An alternative to this "continuation-passing" style, using fields, is discussed below.

Recursion and internal choice. Method `pushN()` illustrates the consumption of a recursive typestate offering a choice. The loop of the form `do-while(exp)` requires `s` to initially be either `Empty` or `NonEmpty`; at each iteration the client decides which of the available methods to call. In this case it chooses to push another value onto the stack. This leaves the stack in state `NonEmpty`, allowing another choice to be made on the next iteration. This is compatible with the recursive structure of the `NonEmpty` state, which permits an unbounded number of `push()` operations, looping back to `NonEmpty` each time.

Recursion and external choice. Method `popAll(Stack s)` also illustrates the consumption of a recursive typestate, but here the stack rather than the client makes the choice. (In session type terminology, the client offers an *external choice*.) This takes the form of a labelled `do-while(true)` in conjunction with a `switch`. The `switch` statement inspects the `Check` enumeration returned by `isEmpty`: in the `NONEMPTY` case, the loop continues, and in the `EMPTY` case the loop terminates. Due to their particular control flow, loops of the form

```
label: do { switch { block } } while(true)
```

are a suitable pattern for consuming a recursive typestate when the condition on the recursion is an external choice (i.e. based on an enumeration label).

Linear objects. Mungo ensures linear usage of objects that follow a typestate protocol; aliasing on objects allows for different method calls on an object that might lead to an inconsistent typestate. Notice that in line 19 of the `StackUser` example:

```
19      s2 = su.popAll(s);
```

the return value of `popAll()` is assigned to `s2`. Now, suppose line 20 were replaced with the following:

```
20      s = su.pushN(s,64);
```

In this case Mungo would report a linearity error on argument `s` in `su.pushN(s, 64)` informing the programmer that variable `s` is used uninitialised, because the usage of variable `s` in line 19 as an argument consumed its linear value.

Inferring typestate for fields. Using fields to store objects can lead to a more idiomatic object-oriented style than explicitly passing values between methods. To illustrate this, we define a second client, `StackUser2`, that stores a `Stack` as a field.

```
1 class StackUser2 {
2   private Stack s;
3   StackUser2() { s = new Stack(); }
4   boolean pushN(int n) {
5     do{ s.push(n--); }
6     while(n>0);
7     return true;}
8   void popAll()
9   { loop : do {
10        System.out.println(s.pop());
11        switch(s.isEmpty()) {
12          case EMPTY: break loop;
13          case NONEMPTY: continue loop;
14      } while(true); }
15  void finish() { s.deallocate(); }
16  public static void main(String[] args)
17  { StackUser2 su = new StackUser2();
18    if(su.pushN(15)||su.pushN(32)) su.pushN(32);
19      su.popAll(); su.finish();
20  } }
```

To track the typestate of a field we need to know the possible sequences in which methods of its containing class may be called, which in turn, requires having a typestate for the containing class. To track the typestate of the field `s`, we need to

provide a typestate for StackUser2. This state machine will then drive typestate checking for those fields of StackUser2 which have their own typestate definitions. For example, we could define the following StackUserProtocol for StackUser2:

```
typestate StackUserProtocol {
  Init = { boolean pushN(int): Cons,
           void finish() : end}
  Cons = { boolean pushN(int): Cons,
           void popAll(): Init} }
```

Typechecking the field s of StackUser2 follows the possible sequences of method calls specified by StackUserProtocol, and also takes into account the constructor body of StackUser2. Then, Mungo can guarantee that if a StackUser2 instance is used according to StackUserProtocol, then the Stack field of the object is also used according to StackProtocol.

Short-circuit boolean expressions. Line 18 in the StackUser2 example above illustrates a final technical detail of typestate inference. The inference algorithm takes into account the fact that logical disjunction short-circuits if the first disjunct evaluates to true. Mungo will ensure that the typestate of su is consistent with there being either one, two or three successive invocations of pushN().

4. Case study: typechecking SMTP

In order to show the practicality and robustness of our StMungo and Mungo toolchain, we have developed a substantial case study in which we statically typecheck an SMTP client. We use this client to communicate with the gmail server. The full source code of the SMTP client can be found in [1].

SMTP (Simple Mail Transfer Protocol) is an Internet standard electronic mail transfer protocol, which typically runs over a TCP (Transmission Control Protocol) connection. We consider the version defined in RFC 5321 [44]. An SMTP interaction consists of an exchange of text-based commands between the client and the server. For example, the client sends the EHLO command to identify itself and open the connection with the server. The commands MAIL FROM: <address> and RCPT TO: <address> specify the e-mail address of the sender and the receiver of the e-mail, respectively. The DATA command allows the client to specify the text of the e-mail. The QUIT command is used to terminate the session and close the connection. The responses from the server have the following format: three digits followed by an optional dash "-", such as 250-, and then some text, like OK. The server might reply to EHLO with 250 <text> or to MAIL FROM or RCPT TO with 250 OK.

To typecheck the SMTP protocol using StMungo and Mungo, we first represent the text-based commands as messages in a Scribble global protocol, based on Hu and Yoshida [28].

```
global protocol SMTP(role S, role C) {
  // Global interaction between server and client.
  _220(String) from S to C;
  rec X1 {
    choice at S { 250dash(String) from S to C; continue X1; }
    or { 250(String) from S to C; ... }
    ... } }
```

Then, we use the Scribble tool to validate and project the above global protocol into local protocols, one for each role. We focus only on the client side and describe the SMTP_C local protocol. This fragment of code of the SMTP describes a loop (rec X1), in which the server S performs a choice between the messages _250DASH and _250. Next, other loops follow (rec Z1 and rec Z3), where in the second one the client C chooses among the messages SUBJECT, to send the subject, DATALINE, to send a line of text, or ATAD to terminate the e-mail by sending a dot.

```
local protocol SMTP_C(role S, self C) {
  _220(String) from S; ...
    rec X1 {
      choice at S { _250dash(String) from S; continue X1; }
      or { _250(String) from S; ...
        rec Z1 {
        ... data(String) to S; ...
        rec Z3 {
          choice at C { subject(String) to S; continue Z3; }
          or{dataline(String) to S;continue Z3;}
          or{atad(String) to S;_250(String) from S;continue Z1;}
        } } }   ...  } }
```

StMungo translates the local protocol (`SMTP_C`) into a typestate specification (`CProtocol`). In addition, it generates a skeletal implementation based on sockets, although other implementations are possible. Every interaction in the local protocol becomes a method call in the typestate specification, as we will see shortly. State definitions group methods into choices and impose sequencing.

Running the StMungo tool on `SMTP_C` produces the following files:

1. `CProtocol.protocol`, which captures the interactions local to the `SMTP_C` role as a typestate specification.
2. `CRole.java`, which is a class that implements `CProtocol` by communication over Java sockets. This is an API that can be used to implement the SMTP client endpoint.
3. `CMain.java`, which is a skeletal implementation of the SMTP client endpoint. It runs as a Java process and provides a `main()` method that uses `CRole` to communicate with the other participants, in this case the SMTP server.

The `CProtocol` generated by StMungo is defined as follows.

```
1 typestate CProtocol {
2   State0 = { String receive_220StringFromS(): State1 }
3   ...
4   State3 = { Choice1 receive_Choice1LabelFromS():
5                          <_250DASH:State4, _250:State5 > }
6   State4 = { String receive_250dashStringFromS(): State3 }
7   State5 = { String receive_250StringFromS(): State6 }
8   ...
9   State27 = { void send_dataStringToS(String): State28 }
10  ...
11  State29 = {void send_SUBJECTToS():State30,
12              void send_DATALINEToS(): State31,
13              void send_ATADToS():State32} ...}
```

The receive and send messages in the `SMTP_C` local protocol are translated as typestate methods in the `CProtocol` typestate specification. For example, the message `_220(String)` received from `S` in line 2 in `SMTP_C` becomes a method with signature:

```
String receive_220StringFromS()
```

given in line 2 in `CProtocol`. Similarly, the message `data(String)` sent to `S` and given in line 7 of `SMTP_C` becomes a method with the following signature:

```
void send_dataStringToS(String)
```

given in line 9 in `CProtocol`.

Let us now comment on choice. The *external* choice made at role `S` different from `self` is given in lines 4–18 of `SMTP_C`. For every external choice in the local protocol there is an enumerated type in the typestate, such as the following:

```
enum Choice1 { _250DASH, _250; }
```

The values of `Choice1` are determined by the first interaction of every branch in the choice. The external choice itself is translated as a receive method returning the enumerated type `Choice1` and given in lines 4–5 of `CProtocol`:

```
State3 = { Choice1 receive_Choice1LabelFromS():
                        <_250DASH:State4, _250:State5 > }
```

After choosing one of the branches, `_250DASH` or `_250`, the payload of type `String` is received via another method call, following `receive_250dashStringFromS()` in line 6 and `receive_250StringFromS()` in line 7, respectively for the available choices.

The *internal* choice made at `self`, namely role `C` (lines 11–17 of `SMTP_C`), is translated into a set of send methods, one for each branch of the choice (lines 12–14 of `CProtocol`). When running the program, only one of these methods will be called, thus performing a single message selection corresponding to it.

`CRole` implements all the methods in `CProtocol`. In this implementation, since communication occurs on Java sockets, we declare and create a new socket to connect to the `gmail` server. This is given in lines 3 and 5, respectively, in `CRole`.

```
1 @Typestate("CProtocol")
2 class CRole {
3   private Socket socketS = null; ...
4   public CRole()
5   { socketS = new Socket("smtp.gmail.com", 587); ...}
6    /* CProtocol  method definitions */ }
```

$$
\begin{array}{lrcl}
\text{Type Decl} & D & ::= & \texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \mid \texttt{enum}\ E\ \{\widetilde{l}\} \\
\text{Typestate} & S & ::= & \widetilde{H} \mid \mu X.S \mid X \\
\text{Signature} & H & ::= & T\ m(T) : S \mid E\ m(T) : \langle S_l \rangle_{l \in E} \\
\text{Field} & F & ::= & T\ f \\
\text{Type} & T & ::= & C \mid E \mid \texttt{bool} \mid \texttt{void} \\
\text{Method} & M & ::= & T\ m(T\ x)\ \{e\} \\
\text{Constant} & c & ::= & l \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{null} \mid * \\
\text{Path} & r & ::= & \texttt{this} \mid r.f \mid x \\
\text{Expr} & e & ::= & c \mid r \mid r.m(e) \mid r.f = e \mid e; e \mid r.f = \texttt{new}\ C \mid \lambda : e \\
 & & & \mid \texttt{continue}\ \lambda \mid \texttt{switch}\ (e)\ \{e_l\}_{l \in E} \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e
\end{array}
$$

Fig. 1. Top-level syntax.

We now describe the correspondence between the text-based commands in SMTP and the method calls in Mungo. Consider "`SUBJECT: Hello World`" which is an atomic command composed of the keyword `SUBJECT` and the subject text. We use an intermediate layer to split this command into two separate method calls, shown in lines 7–9 in the `main` method below. The first, `send_SUBJECTToS()`, selects the command `SUBJECT`. The second, `send_subjectStringToS("Hello World")`, sends the message "`SUBJECT: Hello World`". The intermediate layer is also used when receiving a command from the server. The command name is converted into a value of an enumeration, and the associated text is treated as a separate message.

Finally, `CMain.java` contains the `main` method where the `CRole` object is created and used to implement the client logic.

```
1  public static void main(String[] args) {
2   CRole currentC =  new CRole();
3   ... _Z3:
4   do{ ...
5    switch(/*label to be sent*/) {
6      case /*SUBJECT*/:
7        currentC.send_SUBJECTToS();
8        String subject = // input subject;
9        currentC.send_subjectStringToS (/*subject*/); continue _Z3;
10     case /*DATALINE*/:
11     case /*ATAD*/:
12       currentC.send_ATADToS();
13       currentC.send_atadStringToS (/*single dot*/);
14       String _250msg = currentC.receive_250StringFromS();
15       continue _Z1; }
16   } while(true); }
```

Typically the programmer would flesh out the skeletal implementation with extra logic that, for example, gets relevant input from the user or decides which choice to make when several are available, or customise `CMain` by adding SSL connection code for authentication with the `gmail` server. Mungo is able to statically check `CMain`, or any code that uses a `CRole` object, to ensure that methods of the protocol are called in a valid sequence and that all possible responses are handled. The programmer is not required to use the skeleton implementation of `CMain`, or even the `CRole` API. It is possible to write new code that uses the API, or to use the typestate specification to guide the development of an alternative API, or to refactor the typestate specification itself.

5. A core calculus for Mungo

We define the syntax and operational semantics of a core object-oriented calculus, based on Gay et al. [23], which we use to formalise Mungo. We only formalise the typestate inference system and not the ability of Mungo to work with full Java, as this would require formalising a large subset of Java.

Syntax. The syntax of the calculus is given in Fig. 1. We use $\widetilde{\ }$ to denote a possibly empty set of elements that range over the subject meta-variable. A program is a set of *type declarations* $\widetilde{D}$, each of which declares either a class or an enumerated type. A class declaration defines a *class* named C with typestate specification S, fields $\widetilde{F}$ and methods $\widetilde{M}$. An enumeration declaration defines an *enumerated type* named E with a non-empty set $\widetilde{l}$ of *enum values*. For simplicity, our language has no support for inheritance or interfaces.

We assume as an implicit context a program $\widetilde{D}$, where every class or enumeration has a unique name, and the fields and methods of a given class also have unique names. For any $\texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \in \widetilde{D}$ we write $\mathsf{fields}(C)$ for $\widetilde{F}$, $\mathsf{methods}(C)$ for $\widetilde{M}$, and $\mathsf{typestate}(C)$ for S. For any $\texttt{enum}\ E\ \{\widetilde{l}\} \in \widetilde{D}$ we write $\mathsf{enums}(E)$ for $\widetilde{l}$.

Heap value	o	$::=$	$C[\widetilde{f : o}] \mid c$
Runtime path	r	$::=$	$\texttt{root} \mid r.f$
Expression	e	$::=$	$\ldots \mid e@r$
Value	v	$::=$	$c \mid r$
Typestate	S	$::=$	$\ldots \mid \langle S_l \rangle_{l \in E}$
Typestate action	s	$::=$	$T\ m(T) \mid E\ m(T) : l \mid l$
Context	$\mathcal{E}$	$::=$	$[] \mid r.m(\mathcal{E}) \mid r.f = \mathcal{E} \mid \mathcal{E}; e$
			$\mid \texttt{switch}\,(\mathcal{E})\,\{e_l\}_{l \in E} \mid \mathcal{E}@r \mid \texttt{if}\,(\mathcal{E})\,e\,\texttt{else}\,e$
Action	ℓ	$::=$	$r.f.\texttt{new}\,C \mid r.\langle l \rangle \mid r.T\ m\ T' \mid r.f = v \mid \tau \mid \texttt{if}$

Fig. 2. Runtime syntax.

A typestate definition S specifies a state machine that has as actions the methods of a class. A typestate definition is either an *internal choice* $\widetilde{H}$ of method signatures, or a recursive typestate $\mu X.S$, which may contain the recursive typestate variable X. A method signature H can have two forms, depending on whether the method transitions to a state S, or it is an *external choice* $E\ m(T) : \langle l : S_l \rangle_{l \in E}$ with the method signature defining the transition to one of the possible states $\langle S_l \rangle_{l \in E}$; in the latter case the return type of the method must be E. The empty or *inactive* typestate $\{\}$ can also be written `end`. Well-formedness conditions ensure that state $\mu X.X$ is not well-formed and that all state definitions are closed. Although in the formal calculus typestates are anonymous, in Mungo they have a nominal syntax, as the examples in § 3 showed.

A *type* is either the name of a class or enumeration, `void` or `bool`. A field declaration is a field name f associated with a type T. A method declaration $T\ m(T'\ x)\ \{e\}$ specifies a return type T, the name m of the method, the type T' of the parameter x, and the expression e that comprises the method body. A path is either the atomic path `this` denoting the current object (receiver), the composite path $r.f$ denoting the field f of the object denoted by r, or a parameter x. At runtime paths are resolved to heap locations (see runtime syntax below). A *constant* is the special value `null`, which is assignable to any class type, a literal value `tt` or `ff` of type `bool`, a literal value $*$ of type `void`, or an enum value l.

In the expression forms method call $r.m(e)$, field assignment $r.f = e$, and object creation $r.f = \texttt{new}\ C$, the target object of the invocation or assignment is restricted to a path r, rather than an arbitrary expression. This is to allow the typestate of the target to be tracked by the type system. The other expression forms include constants, paths, sequential composition $e; e'$, `switch` expressions, `if` ...`else` expressions, labelled expressions $\lambda : e$, and `continue` expressions that jump to the enclosing expression labelled by λ.

Configurations and runtime syntax. Fig. 2 extends the source syntax with additional runtime constructs used by the operational semantics.

A *configuration* h, e is a pair of a *heap* h and *runtime expression* e. The heap h is defined as an *object* $C[\widetilde{f : o}]$, where C is the class of the object and $\widetilde{f : o}$ are its fields; the contents o of each field is either a constant c or another object. The "heap" is a tree of objects, with neither cycles nor sharing, due to the linearity of object references enforced by the type system (see § 6).

The expression e in a configuration h, e is a *runtime expression* in which every (compile-time) path of the form `this`, $r.f$ or x has been replaced by a *runtime path* that refers to a heap value. A runtime path r in a heap h is either the atomic path `root` denoting h itself or the composite path $r'.f$ denoting the field f of the object denoted by r', where r' is also a path in h. Runtime expressions also include the form $e@r$, which is an expression e that has been tagged with $@r$ to track the active receiver. A *value* v is either a constant c or runtime path r. Every runtime expression is either a value, or uniquely of the form $\mathcal{E}[e]$, where $\mathcal{E}$ is an *evaluation context* (an expression with a hole). As usual, the notation $\mathcal{E}[e]$ denotes the plugging of the hole in $\mathcal{E}$ with an expression e.

The operational semantics is annotated with labels ℓ that denote the creation of a new object ($r.f.\texttt{new}\ C$), an enum value choice ($r.\langle l \rangle$), method call ($r.T\ m\ T'$), assigning a field ($r.f = v$), the conditional label (`if`), and the silent label (τ). The definition of states is extended to the set of enum values $\langle l : S_l \rangle_{l \in E}$ and we define action labels s for labels: internal choice $T\ m(T)$, external choice $E\ m(T) : l$, and for enum values l.

Labelled reduction semantics. We define heap access and update functions that are used by the reduction relation in Fig. 3:

$$\left.\begin{array}{ll} h(\texttt{root}) & = h \\ h(r.f) & = o \\ h\{r.f \mapsto o'\} & = h\{r \mapsto C[\widetilde{f : o}, f : o']\} \end{array}\right\} \text{if} \quad h(r) = C[\widetilde{f : o}, f : o]$$

The root object is accessed via $h(\texttt{root})$. The access of a field $h(r.f)$ is inductively defined on the access of $h(r)$. Similarly, we use the heap access function to update object fields as in $h\{r.f \mapsto o\}$.

Fig. 3 defines the labelled reduction semantics; hereafter by "expression" we shall mean runtime expression, and by "path" runtime path, unless otherwise indicated. Rule R-Ctx lifts the semantic rules to an arbitrary expression using an evaluation context. Rule R-Seq discards the value v in a τ label and proceeds with the evaluation of e. Rules R-True and R-False are the usual rules for the `if` ...`else` expression and are annotated with label `if`. Rule R-New is labelled with $r.f.\texttt{new}\ C$ and overwrites the contents of the field $r.f$ by a new object $C[\widetilde{f = \mathsf{init}(T)}]$ whose fields are all initialised to the value $\mathsf{init}(T)$, where T is the type of the field, defined by:

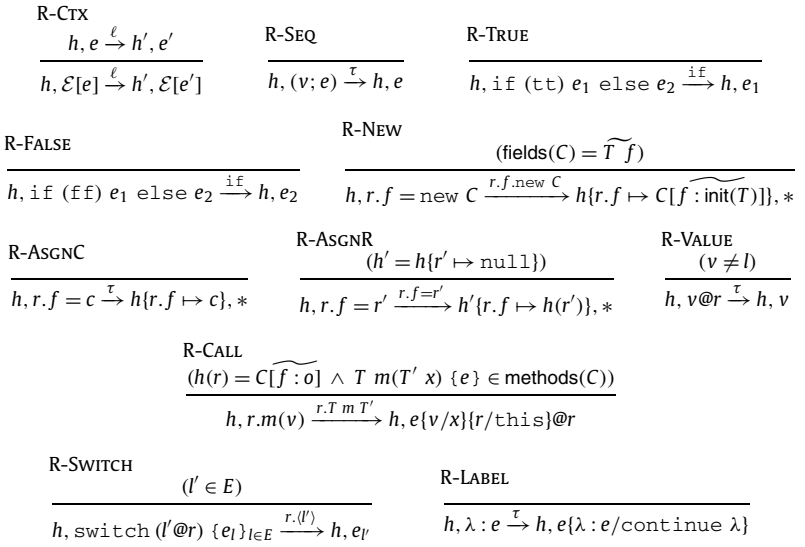


Fig. 3. Operational semantics.

$$\begin{array}{llll} \mathsf{init}(C) & = \texttt{null} & \mathsf{init}(E) & = E_{\mathsf{init}} \\ \mathsf{init}(\mathsf{bool}) & = \texttt{ff} & \mathsf{init}(\texttt{void}) & = * \end{array}$$

where for every enumerated type E we require there to be a distinguished element $E_{\mathsf{init}} \in \mathsf{enums}(E)$. The result of R-New is the `void` value $*$. The object is constructed at a location within an already existing object $r.f$. There are two assignment rules, depending on whether the value being assigned is a constant or an object path. Both forms return the `void` value $*$. A constant c has no associated typestate and may be used unrestrictedly; therefore the R-AsgnC rule is labelled with τ and simply updates the heap to store c in $r.f$. A path r', on the other hand, refers to an object and must be used linearly. Therefore the effect of the R-AsgnR rule is to *relocate* the object from r' to $h.r$, leaving `null` at its old location. The annotation label for R-AsgnR is $r.f = r'$. Although any existing object at $h.r$ will be overwritten, the type system (§6) only permits the assignment if the typestate of that object is fully consumed.

The R-Call rule is labelled with $r.T\ m\ T'$ and resolves the method m by first looking up the receiver r in the heap, which must be an object $C[\widetilde{f : o}]$, and then selecting the method m from the definition of C. Prior to executing the selected method, we convert its body e, which is a source-level expression, into a runtime expression by substituting the runtime path r for `this` and also v for the formal parameter. In addition, the resulting runtime expression is tagged with $@r$, recording the fact that r is the active receiver. The active receiver tag $@r$ on a value is removed using a τ label when the value is fully evaluated and it is not an enum label, as defined by rule R-Value. If the value returned by the method is an enum label l', then it must occur as the scrutinee of a `switch` expression; rule R-Switch defines the reduction via action $r.\langle l' \rangle$, of the `switch` expression to the branch indicated by l'. The r is used in the reduction label to indicate which object made the choice. Rule R-Label is labelled with τ, and says that a labelled expression $\lambda : e$ discards the λ and substitutes a copy of the labelled expression for every occurrence of `continue` λ that occurs in the loop body e.

It is easy to show that the operational semantics is deterministic. Assume a heap consisting of an instance of class C, where given $\mathsf{fields}(C) = \widetilde{T\ f}$, each field of C is initialised with the corresponding value $\mathsf{init}(T)$. Execution can then be initiated using a top-level expression that substitutes path `this` with path `root`.

6. Typestate inference

In this section we formalise the core of Mungo's typestate inference system and prove its safety properties. The formalisation makes a simplification, which is that the body of a method is analysed every time the method is called. If the implementation of Mungo used this simplification, then as well as being inefficient it would be impossible to analyse recursive methods. Section 6.4 informally explains how Mungo uses the concept of *partial typestate* to remove this restriction.

The typestate inference system infers a *typestate specification* for a class, consistent with the static usage of instances of the class; the inferred typestate is then checked against the declared typestate of the class. Proving the soundness of the inference system requires proving that the trace of the execution of a well-typed program is included in the trace of the inferred type for that program. A sound inference system should be able to guarantee the progress property requiring that a program either reduces or is a value. The syntax of the inferred types, ranged over by U, and the typing context, ranged over by Δ, are defined below:

$$U \ ::= \ C[S] \mid E \mid \texttt{bool} \mid \texttt{void} \mid \texttt{bot} \quad \Delta \ ::= \ \emptyset \mid \Delta, r : U \mid \Delta, \lambda : X$$

$$\text{S-Start}\ \frac{\emptyset \vdash S \preccurlyeq_{\text{sbt}} S'}{S \preccurlyeq_{\text{sbt}} S'} \qquad \text{S-End}\ \frac{}{\mathcal{R} \vdash \texttt{end} \preccurlyeq_{\text{sbt}} \texttt{end}}$$

$$\text{S-Terminate}\ \frac{(S, S') \in \mathcal{R}}{\mathcal{R} \vdash S \preccurlyeq_{\text{sbt}} S'} \qquad \text{S-Method}\ \frac{\mathcal{R} \vdash S \preccurlyeq_{\text{sbt}} S'}{\mathcal{R} \vdash T'\ m(T) : S \preccurlyeq_{\text{sbt}} T'\ m(T) : S'}$$

$$\text{S-Rec1}\ \frac{\mathcal{R} \cup \{(S, \mu X.S')\} \vdash S \preccurlyeq_{\text{sbt}} S'\{\mu X.S'/X\}}{\mathcal{R} \vdash S \preccurlyeq_{\text{sbt}} \mu X.S'}$$

$$\text{S-Set}\ \frac{\widetilde{H} \neq \emptyset \qquad \forall H \in \widetilde{H}, \exists H' \in \widetilde{H'}.\ \mathcal{R} \vdash H \preccurlyeq_{\text{sbt}} H'}{\mathcal{R} \vdash \widetilde{H} \preccurlyeq_{\text{sbt}} \widetilde{H'}}$$

$$\text{S-Enum}\ \frac{\forall l \in E.\ \mathcal{R} \vdash S_l \preccurlyeq_{\text{sbt}} S'_l}{\mathcal{R} \vdash E\ m(T) : \langle S_l \rangle_{l \in E} \preccurlyeq_{\text{sbt}} E\ m(T) : \langle S'_l \rangle_{l \in E}} \qquad \text{S-Class}\ \frac{S \preccurlyeq_{\text{sbt}} S'}{C[S] \preccurlyeq_{\text{sbt}} C[S']}$$

$$\text{S-Bot}\ \frac{}{\texttt{bot} \preccurlyeq_{\text{sbt}} U} \qquad \text{S-Grnd}\ \frac{U \in \{E, \texttt{bool}, \texttt{void}\}}{U \preccurlyeq_{\text{sbt}} U} \qquad \text{S-Empty}\ \frac{}{\emptyset \preccurlyeq_{\text{sbt}} \Delta}$$

$$\text{S-Delta}\ \frac{\Delta \preccurlyeq_{\text{sbt}} \Delta' \qquad U \preccurlyeq_{\text{sbt}} U'}{\Delta, r : U \preccurlyeq_{\text{sbt}} \Delta', r : U'} \qquad \text{S-Lambda}\ \frac{\Delta \preccurlyeq_{\text{sbt}} \Delta'}{\Delta, \lambda : X \preccurlyeq_{\text{sbt}} \Delta', \lambda : X}$$

Fig. 4. Subtyping relation (symmetric rule S-Rec2 omitted).

The inferred types U differ from top-level types T in that every class type C is refined with a typestate specification S. There is a distinguished bottom type `bot`, which is used to type `continue` statements. It use will be illustrated when we discuss recursion and choice, later in this section. Typing context Δ is a partial function from runtime paths r to types U, and from expression labels λ to recursive type variables X. A type U that is not a class type is referred to as a *constant type*.

The inference system uses a subtyping relation $\preccurlyeq_{\text{sbt}}$ and a binary operator $\text{join}(\cdot, \cdot)$.

Definition 2. ($\preccurlyeq_{\text{sbt}}$, $=_{\text{sbt}}$, join) The following relations are defined on typestates, inferred types and typing contexts.

- The *subtyping* relation $\preccurlyeq_{\text{sbt}}$ is defined by the rules in Fig. 4.
- The *equivalence* relation is defined as $=_{\text{sbt}} = \preccurlyeq_{\text{sbt}} \cap \preccurlyeq_{\text{sbt}}^{-1}$.
- The *join* operator $\text{join}(\cdot, \cdot)$ is defined by the rules in Fig. 5.

Subtyping on typestates is essentially a simulation preorder and is given in an algorithmic style [40, Chapter 21]. It constructs a set $\mathcal{R}$ of pairs of typestates using rules S-Rec1 and S-Rec2. The algorithm terminates either when `end` matches `end` (rule S-End) or when a pair of typestates is revisited (rule S-Terminate). Rule S-Method checks for prefix-matching. Rule S-Set requires covariance on subtyping with the empty set being treated as a special case. Rule S-Enum matches the external choice prefix. It requires subtyping on typestates for every value of the enumerated type. By analogy with subtyping on session types [21], it should be possible to generalise the definition to allow contravariant subtyping in the set of enumerated values. This might be relevant if we wanted to support inheritance, but it is not necessary for the case studies that we have implemented so far.

The subtyping relation extends to inferred types and typing contexts. Proving that it is a preorder is standard; we use reflexivity and transitivity in proofs anout the type system.

The join operator computes an upper bound with respect to $\preccurlyeq_{\text{sbt}}$. It is used to compute a common typestate in typing rules that combine multiple execution paths. The most interesting case of join on typestates is the join of method signatures. For methods common to the two typestates, the continuation typestates are joined; the remaining methods are combined using set union. A join involving a recursive typestate is defined by unfolding. The operation extends to inferred types U (in particular, to types of the form $C[S]$) and typing contexts. Finally, we define a transition relation on typestates as follows.

Definition 3. (*Transition on typestates*) The transition relation $S \xrightarrow{s} S'$ is defined by the following rules:

$$T\ m(T) : S \xrightarrow{T\ m(T)} S \qquad \frac{l' \in \text{enums}(E)}{E\ m(T) : \langle S_l \rangle_{l \in E} \xrightarrow{E\ m(T):l'} S_{l'}}$$

$$\frac{H \in \widetilde{H} \qquad H \xrightarrow{s} S}{\widetilde{H} \xrightarrow{s} S} \qquad \frac{S\{\mu X.S/X\} \xrightarrow{s} S'}{\mu X.S \xrightarrow{s} S'} \qquad \frac{l' \in \text{enums}(E)}{\langle S_l \rangle_{l \in E} \xrightarrow{l'} S_{l'}}$$

$$\mathsf{join}(T\ m(T') : S, \widetilde{H}) = \begin{cases} \{T\ m(T') : \mathsf{join}(S, S')\} \cup \widetilde{H}' & \text{if } \widetilde{H} = \{T\ m(T') : S'\} \cup \widetilde{H}' \\ \{T\ m(T') : S\} \cup \widetilde{H} & \text{otherwise} \end{cases}$$

$$\mathsf{join}(E\ m(T) : \langle S_l \rangle_{l \in E}, \widetilde{H}) = \begin{cases} \{E\ m(T) : \langle l : \mathsf{join}(S_l, S'_l) \rangle_{l \in E}\} \cup \widetilde{H}' & \text{if } \widetilde{H} = \{E\ m(T) : \langle S'_l \rangle_{l \in E}\} \cup \widetilde{H}' \\ E\ m(T) : \langle S_l \rangle_{l \in E} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathsf{join}(\{H\} \cup \widetilde{H}, \widetilde{H}') &= \mathsf{join}(\widetilde{H}, \mathsf{join}(H, \widetilde{H}')) \\ \mathsf{join}(\mathtt{end}, \mathtt{end}) &= \mathtt{end} \\ \mathsf{join}(\mu X.S_1, S_2) &= \mathsf{join}(S_1\{\mu X.S_1/X\}, S_2) \\ \mathsf{join}(C[S], C[S']) &= C[\mathsf{join}(S, S')] \\ \mathsf{join}(U, \mathtt{bot}) &= U \\ \mathsf{join}(\mathtt{bot}, U) &= U \\ \mathsf{join}(U, U) &= U \qquad (U \in \{E, \mathtt{bool}, \mathtt{void}\}) \\ \mathsf{join}(\Delta, \Delta') &= \{r : C[\mathsf{join}(S, S')] \mid r : C[S] \in \Delta, \\ &\qquad r : C[S'] \in \Delta'\} \cup (\Delta \setminus \Delta') \cup (\Delta' \setminus \Delta) \end{aligned}$$

Fig. 5. Join operator (symmetric recursion rule omitted).

The first two rules state that a method-prefixed typestate reduces to its continuation, using the method signature as the label. The next two rules are contextual, stating that reduction can occur in a set of typestates and under recursion, respectively. The last rule defines a reduction on a runtime typestate, as defined in Fig. 2. It states that a branching typestate reduces to one of its components, using the corresponding enumerated value as the label. Later we will use the notation $S \xrightarrow{\vec{s}} S'$ for a sequence of transitions, where $\vec{s}$ is a sequence of method signatures and labels.

6.1. Typestate inference rules

Before introducing the inference rules, we give the form of the judgements:

$$\Delta \vdash e : U \dashv \Delta' \qquad \Delta \vdash C[S] \qquad \vdash \mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \qquad \vdash \widetilde{D}$$

The first one is the typing judgement for expressions. The judgement is read from right to left. It takes as input the typing context Δ' and the expression e, and algorithmically computes the type U. The effects of the expression on Δ' are then captured in Δ. (However, it is interesting to note that the judgement can also be read from left to right in a type system fashion, where the expression "consumes" Δ in order to produce Δ'.) The second judgement infers the typestates of the fields of a class when the class is used according to its declared typestate. The last two typing judgements state the well-formedness of classes and programs.

The typestate inference rules for expressions are given in Fig. 6. The rules are syntax-directed, meaning that at any point in the derivation there is only one rule which is applicable. Rules Void, Bool, Enum and Null type the constants with their corresponding types under any typing context without producing any effect on it, namely the left and right typing contexts are the same. Rules Strengthen and Weaken allow arbitrary removal and addition, respectively, of inactive (end) typestate assumptions.

Typestate linearity. The typestate inference system uses linearity to forbid aliasing. The following example explains the relevant rules: Seq, PathR, PathC, AsgnR, AsgnC, and New. Consider the following code using the class `Stack` defined in §3:

```
s = new Stack; k = s        (1)
```

The expression matches rule Seq. Suppose

$$\Delta_0 = \mathtt{s} : \mathtt{Stack}[\mathtt{end}], \mathtt{k} : \mathtt{Stack}[S]$$

is the input typing context where $S \preccurlyeq_{\mathsf{sbt}} \mathtt{StackProtocol}$. Rule Seq processes the second expression before the first, because the output typing context of the second expression is an input for the first. To type the second expression, we use AsgnR which in turn requires a typestate for `s`. The derivation is:

$$\dfrac{\dfrac{}{\Delta_2 \vdash \mathtt{s} : \mathtt{Stack}[S] \dashv \Delta_1}\ \textsc{PathR}}{\Delta_2 \vdash \mathtt{k = s} : \mathtt{void} \dashv \Delta_0}\ \textsc{AsgnR}$$

$$\textsc{Void}\ \frac{}{\Delta \vdash * : \texttt{void} \dashv \Delta} \qquad \textsc{Bool}\ \frac{}{\Delta \vdash \texttt{tt}, \texttt{ff} : \texttt{bool} \dashv \Delta} \qquad \textsc{Enum}\ \frac{l \in \mathsf{enums}(E)}{\Delta \vdash l : E \dashv \Delta}$$

$$\textsc{Null}\ \frac{\texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \in \widetilde{D}}{\Delta \vdash \texttt{null} : C[\texttt{end}] \dashv \Delta} \qquad \frac{\Delta \vdash e : U \dashv \Delta' \qquad r \notin dom(\Delta')}{\Delta \vdash e : U \dashv \Delta', r : C[\texttt{end}]}\ \textsc{Weaken}$$

$$\textsc{Strengthen}\ \frac{\Delta, r : C[\texttt{end}] \vdash e : U \dashv \Delta'}{\Delta \vdash e : U \dashv \Delta'} \qquad \frac{U \neq C[S]}{\Delta, r : U \vdash r : U \dashv \Delta, r : U}\ \textsc{PathC}$$

$$\textsc{PathR}\ \frac{r \neq \texttt{this}}{\Delta, r : C[S] \vdash r : C[S] \dashv \Delta, r : C[\texttt{end}]} \qquad \frac{\Delta \vdash e : U \dashv \Delta' \qquad \Delta =_{\mathsf{sbt}} \Delta''}{\Delta'' \vdash e : U \dashv \Delta'}\ \textsc{Equiv}$$

$$\textsc{AsgnC}\ \frac{U \neq C[S] \quad \Delta \vdash e : U \dashv \Delta', r : U}{\Delta \vdash r = e : \texttt{void} \dashv \Delta', r : U} \qquad \frac{r \neq \texttt{this} \quad \Delta \vdash e : C[S] \dashv \Delta', r : C[\texttt{end}]}{\Delta \vdash r = e : \texttt{void} \dashv \Delta', r : C[S]}\ \textsc{AsgnR}$$

$$\textsc{New}\ \frac{r \neq \texttt{this} \qquad S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \qquad \forall r.f : C'[S'] \in \Delta \implies S' = \mathsf{end}}{\Delta, r : C[\texttt{end}] \vdash r = \texttt{new}\ C : \texttt{void} \dashv \Delta, r : C[S]} \qquad \frac{\Delta \vdash e_1 : U' \dashv \Delta'' \quad \Delta'' \vdash e_2 : U \dashv \Delta' \quad U' \neq C[S] \qquad U' \neq \texttt{bot}}{\Delta \vdash e_1; e_2 : U \dashv \Delta'}\ \textsc{Seq}$$

$$\frac{T\ m(T'\ x)\ \{e'\} \in \mathsf{methods}(C) \quad S' =_{\mathsf{sbt}} S \quad \Delta'', r : C[S'], x : U' \vdash e'\{r/\texttt{this}\} : U \dashv \Delta', r : C[S], x : \mathsf{initT}(T') \quad \Delta \vdash e : U' \dashv \Delta'', r : C[\{T\ m(T') : S\}]}{\Delta \vdash r.m(e) : U \dashv \Delta', r : C[S]}\ \textsc{Call}$$

$$\textsc{Switch}\ \frac{\forall l \in E.\ \Delta_l, r : C[S_l] \vdash e_l : U_l \dashv \Delta' \quad \Delta \vdash r.m(e) : E \dashv \Delta'' \quad \Delta'' = \mathsf{join}(\{\Delta_l\}_{l \in E})}{\Delta \vdash \texttt{switch}\ (r.m(e))\ \{e_l\}_{l \in E} : \mathsf{join}(\{U_l\}_{l \in E}) \dashv \Delta'} \qquad \frac{\Delta_1 \vdash e_1 : U_1 \dashv \Delta' \quad \Delta_2 \vdash e_2 : U_2 \dashv \Delta' \quad \Delta'' = \mathsf{join}(\Delta_1, \Delta_2) \quad \Delta \vdash e : \texttt{bool} \dashv \Delta''}{\Delta \vdash \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 : \mathsf{join}(U_1, U_2) \dashv \Delta'}\ \textsc{If}$$

$$\textsc{LExpr}\ \frac{\Delta'' \vdash e : U \dashv \Delta', \lambda : X \qquad X\ \text{fresh} \quad \Delta = \{r : C[\mu X.S] \mid r : C[S] \in \Delta''\} \cup \{r : U' \mid r : U' \in \Delta''\ \text{and}\ U' \neq C'[S']\}}{\Delta \vdash \lambda : e : U \dashv \Delta'} \qquad \frac{\Delta = \{r : C[X] \mid r : C[S] \in \Delta'\} \cup \{r : U \mid r : U \in \Delta'\ \text{and}\ U \neq C'[S']\}}{\Delta \vdash \texttt{continue}\ \lambda : \texttt{bot} \dashv \Delta', \lambda : X}\ \textsc{Continue}$$

Fig. 6. Typestate inference rules for expressions.

where $\Delta_0 = \texttt{s} : \texttt{Stack}[\texttt{end}], \texttt{k} : \texttt{Stack}[S]$, $\Delta_1 = \texttt{s} : \texttt{Stack}[\texttt{end}], \texttt{k} : \texttt{Stack}[\texttt{end}]$ and $\Delta_2 = \texttt{s} : \texttt{Stack}[S], \texttt{k} : \texttt{Stack}[\texttt{end}]$. The output typing context, Δ_2, for PathR states that `k` has an `end` typeWstate before the assignment. Rule PathR "guesses" a type for a path expression. However, the combination of PathR and AsgnR enforces a match on the type of `s` in the output typing context Δ_2 and the type of `k` in the input typing context Δ_0. For the first expression in (1) we use rule New. By assumption we satisfy its premise; we have $S \preccurlyeq_{\mathsf{sbt}} \texttt{StackProtocol}$, meaning path `s` is used according to the `StackProtocol` typestate. Rule New always infers `void`; this satisfies the premise of rule Seq, which requires the type of the first expression to be neither a class type (so that discarding does not violate linearity) nor `bot` (to forbid dead code after a `continue` λ expression – see rule Continue). The type of the sequential expression is the type of the second expression, `void`. To summarise the derivations described so far:

$$\frac{\dfrac{S \preccurlyeq_{\mathsf{sbt}} \texttt{StackProtocol}}{\Delta_3 \vdash \texttt{s = new Stack} : \texttt{void} \dashv \Delta_2}\ \textsc{New} \qquad \dfrac{\ldots}{\Delta_2 \vdash \texttt{k = s} : \texttt{void} \dashv \Delta_0}\ \textsc{AsgnR}}{\Delta_3 \vdash \texttt{s = new Stack; k = s} : \texttt{void} \dashv \Delta_0}\ \textsc{Seq}$$

where $\Delta_3 = \texttt{s} : \texttt{Stack}[\texttt{end}], \texttt{k} : \texttt{Stack}[\texttt{end}]$.

To preserve linearity, `s` and `k` exchange their typestates before and after assignment. If the type of `s` in Δ_0 were not `end`, path `s` would still be usable after being read from, violating linearity, as in:

```
s = new Stack; k = s; s.push(5)        (3)
```

Note that, in rules AsgnR and New, the path `this` cannot be assigned to, otherwise the current object would be overwritten in the heap. In rule PathR, the path `this` cannot be read from, because using `this` in an expression would violate the linearity condition that the unique reference to the object is at the call site.

The other rules for paths and assignments are as follows. Rule PathC infers a constant type U for a path r and has no effect in the input typing context, if r is mapped to U in the input typing context. Rule AsgnC is similar to AsgnR, except that e has a constant type U that is left unchanged in the input and output typing contexts.

Loops, choice and recursive typestate. We now explain how a loop can be controlled by the enumerated value returned by a method, and how this leads to the inference of a recursive typestate specification. The example illustrates rules LExpr, Continue, Switch, and If. Consider the following class `StackUser` that defines methods that use a `Stack`. Note that the example is expressed in the core calculus, using Java-style formatting, so there is no `return` keyword.

```
1 class StackUser {
2   Stack pushVal(Stack x) { x.push(2); x }
3   Stack popAll(Stack x)
4   {loop:switch(x.isEmpty())
5         {case EMPTY:x,
6          case NOTEMPTY:x.pop(); continue loop}}}
```

and suppose the input typing context:

$$\Delta_0 = \mathtt{x} : \mathtt{Stack}[\mathtt{end}], \mathtt{this} : \mathtt{StackUser}[\mathtt{end}]$$

The body of method `popAll` in line 4 is a labelled expression, and so rule LExpr applies. The premise computes a typestate for the `switch` expression, using an input context Δ_0 augmented with the assumption $\mathtt{loop} : X$, where X is fresh. Let $\Delta_1 = \Delta_0, \mathtt{loop} : X$. LExpr closes all free occurrences of X in the output typing context. For the `switch` expression, rule Switch computes a typestate for each branch, using the typing context for the entire `switch` expression, Δ_1, as the input context. The inferred output contexts of the branches are then joined and used as an input to infer a typestate for the method call expression which is the condition of the `switch`. The condition should have an enumeration type that matches the labels of the `switch` branches. Finally, the type of the `switch` is the join of the types of its branches. For the `EMPTY` branch we use rule PathR:

$$\textsc{PathR}\ \frac{}{\Delta_2 \vdash \mathtt{x} : \mathtt{Stack}[S] \dashv \Delta_1}$$

where $\Delta_2 = \mathtt{x} : \mathtt{Stack}[S], \mathtt{this} : \mathtt{StackUser}[\mathtt{end}], \mathtt{loop} : X$. For the `NONEMPTY` branch we first use rule Seq, and then rule Continue to infer the typestate of the `continue loop` expression. Continue requires `loop` to be mapped to a recursive variable X in the input typing context. It outputs a typing context where all paths mapped to a typestate are updated to the typestate X:

$$\textsc{Continue}\ \frac{}{\Delta_3 \vdash \mathtt{continue\ loop} : \mathtt{bot} \dashv \Delta_1}$$

where $\Delta_3 = \mathtt{x} : \mathtt{Stack}[X], \mathtt{this} : \mathtt{StackUser}[X]$. The type of a `continue` expression is always `bot`, and so can always be joined with the type of another branch (cf. Fig. 5). To complete the typing of the `NONEMPTY` branch, we apply rule Call for `x.pop()` and conclude with rule Seq. The output typing context is:

$$\Delta_4 = \mathtt{x} : \mathtt{Stack}[\{\mathtt{int\ pop}() : X\}], \mathtt{this} : \mathtt{StackUser}[X]$$

We join the output typing contexts Δ_2 and Δ_4 of the `EMPTY` and `NONEMPTY` branches, and use the result as an input typing context for the method call `x.isEmpty()`, as per the second premise of Switch. The output typing context of Switch is:

$$\Delta_5 \ = \ \mathtt{x} : \mathtt{Stack}[\{\mathtt{Choice\ isEmpty}() : \mathsf{join}(S, \mathtt{int\ pop}() : X)\}], \mathtt{this} : \mathtt{StackUser}[\mathsf{join}(\mathtt{end}, X)]$$

To complete the derivation for LExpr we close the recursive variable X in Δ_5 and obtain the output typing context for the labelled expression in lines 4–5, namely:

$$\Delta_6 \ = \ \mathtt{x} : \mathtt{Stack}[\mu X.\mathtt{Choice\ isEmpty}() : \mathsf{join}(S, \mathtt{int\ pop}() : X)], \mathtt{this} : \mathtt{StackUser}[\mu X.\mathsf{join}(\mathtt{end}, X)]$$

Notice the equivalence of the type $\mu X.\mathsf{join}(\mathtt{end}, X)$, that Δ_6 assigns to path `this`, and the type `end`, meaning that rule Equiv can be applied.

Rule If is similar to Switch. Both conditional branches are individually inferred and then joined to obtain the output typing context of the conditional expression. We further require that the condition has type `bool`.

Method Call. Rule Call records the method call trace of paths in a program, to respect the principle that the trace of the execution of an object follows its inferred typestate. It uses the function $\mathsf{init}\mathsf{T}(\cdot)$, defined by $T \neq C \implies \mathsf{init}\mathsf{T}(T) = T$ and $\mathsf{init}\mathsf{T}(C) = C[\mathtt{end}]$. Rule Call typechecks the method body every time a method is called. This is a simplification for presentational purposes; an algorithm directly extracted from the rules will be unable to construct a type for a recursive method call. However, the rules can be used to derive typings if suitable pre- and post-conditions are put into the derivation

$$\textsc{Method-St}\ \frac{\Delta' \vdash C[S] \quad T_1\ m(T_2\ x)\ \{e\} \in \mathsf{methods}(C) \quad S =_{\mathsf{sbt}} S' \quad \mathsf{refines}(U_1, T_1) \quad \mathsf{refines}(U_2, T_2) \quad \Delta, \mathtt{this} : C[S'], x : U_2 \vdash e : U_1 \dashv \Delta', \mathtt{this} : C[S], x : \mathsf{initT}(T')}{\Delta \vdash C[\{T\ m(T') : S\}]}$$

$$\textsc{Enum-St}\ \frac{\forall l \in E.\ \Delta_l \vdash C[S_l] \quad E\ m(T\ x)\ \{e\} \in \mathsf{methods}(C) \quad \Delta, x : C[S'] \vdash E\ m(T\ x)\ \{e\} : E \dashv \Delta'' \quad \Delta'' = \mathsf{join}(\{\Delta_l\}_{l \in E})}{\Delta \vdash C[E\ m(T) : \langle S_l \rangle_{l \in E}]}$$

$$\textsc{Set-St}\ \frac{\forall H \in \widetilde{H}.\ \Delta_H \vdash C[\{H\}] \quad \Delta = \mathsf{join}(\{\Delta_H\}_{H \in \widetilde{H}})}{\Delta \vdash C[\widetilde{H}]} \qquad \textsc{End-St}\ \frac{\Delta = \{f : C'[\mathtt{end}] \mid C'\ f \in \mathsf{fields}(C)\} \cup \{f : T \mid T\ f \in \mathsf{fields}(C) \text{ and } T \neq C\}}{\Delta \vdash C[\mathtt{end}]}$$

$$\textsc{Rec-St}\ \frac{\Delta' \vdash C[S] \quad \Delta = \{r : C[\mu X.S] \mid r : C[S] \in \Delta'\} \cup \{r : U \mid r : U \in \Delta' \text{ and } U \neq C'[S']\}}{\Delta \vdash C[\mu X.S]}$$

$$\textsc{Var-St}\ \frac{\Delta = \{f : C'[X] \mid C'\ f \in \mathsf{fields}(C)\} \cup \{f : T \mid T\ f \in \mathsf{fields}(C) \text{ and } T \neq C\}}{\Delta \vdash C[X]} \qquad \textsc{Class}\ \frac{\Delta \vdash C[S] \quad \forall f : C'[S] \in \Delta \implies S = \mathtt{end}}{\vdash \mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\}}$$

$$\textsc{Program}\ \frac{\forall D \in \widetilde{D}.D = \mathtt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \implies \vdash D}{\vdash \widetilde{D}}$$

Fig. 7. Typestate inference rules for methods, fields, classes and programs.

by hand. The implementation of Mungo's type inference system uses a more complex notion of partial typestate so that method bodies do not need to be checked at every call site; recursive methods are also supported.

Rule Call includes the hypothesis $S' =_{\mathsf{sbt}} S$. This condition means that the typestate of `this` is preserved by the body of the method. The same condition appears in rule Method-St in Fig. 7. To satisfy this condition, self-called methods must not change the typestate.

As an example of the rule Call, consider the following code that uses class `StackUser`:

```
s = c.pushVal(s)
```

where the input typing context is:

$$\Delta_0 = \mathtt{s} : \mathtt{Stack}[S], \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack}\ \mathtt{popAll}(\mathtt{Stack}) : \mathtt{end}\}]$$

By applying rule AsgnR on the above assignment with input Δ_0, the output typing context in the premise of the rule is:

$$\Delta_1 = \mathtt{s} : \mathtt{Stack}[\mathtt{end}], \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack}\ \mathtt{popAll}(\mathtt{Stack}) : \mathtt{end}\}]$$

We can now apply rule Call on `c.pushVal(s)` as follows:

$$\textsc{Call}\ \frac{\begin{array}{ll}\mathtt{Stack}\ \mathtt{pushVal}(\mathtt{Stack}\ \mathtt{x})\ \{\mathtt{x.push}(2); \mathtt{x}\} \in \mathsf{methods}(\mathtt{StackUser}) & (1)\\ \Delta_2 \vdash (\mathtt{x.push}(2); \mathtt{x})\{\mathtt{c/this}\} : \mathtt{Stack}[S] \dashv \Delta_1 & (2)\\ \Delta \vdash \mathtt{s} : \mathtt{Stack}[\{\mathtt{void}\ \mathtt{push}(\mathtt{int}) : S\}] \dashv \Delta_3 & (3)\end{array}}{\Delta \vdash \mathtt{c.pushVal(s)} : \mathtt{Stack}[S] \dashv \Delta_1}$$

Premise (1) looks up the definition of method `pushVal` in the class of the receiver, `StackUser`. Premise (2) infers a typestate for the method body in which c has been substituted for the keyword `this`. Both the method call and the body use the same input typing context Δ_1. The output typing context from the method body must contain a typestate assumption for the method parameter and receiver:

$$\Delta_2 = \Delta_1, \mathtt{x} : \mathtt{Stack}[\{\mathtt{void}\ \mathtt{push}(\mathtt{int}) : S\}]$$

Then, premise (3) requires a typestate inference in order to match the typestate of the method parameter with the type of the argument s. For this, rule PathR is used where Δ_3 also updates the type of the receiver:

$$\begin{array}{ll}\Delta_3 & = \mathtt{s} : \mathtt{Stack}[\mathtt{end}], \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack}\ \mathtt{pushN}(\mathtt{Stack}) : \{\mathtt{Stack}\ \mathtt{popAll}(\mathtt{Stack}) : \mathtt{end}\}\}]\\ \Delta & = \mathtt{Stack}[\{\mathtt{void}\ \mathtt{push}(\mathtt{int}) : S\}], \mathtt{c} : \mathtt{StackUser}[\{\mathtt{Stack}\ \mathtt{pushVal}(\mathtt{Stack}) : \\ & \qquad \{\mathtt{Stack}\ \mathtt{popAll}(\mathtt{Stack}) : \mathtt{end}\}\}]\end{array}$$

$$\textsc{Switch-AtR}\ \frac{\forall l \in E.\ \Delta_l, r : C[S_l] \vdash e_l : U_l \dashv \Delta' \qquad \Delta \vdash e : E \dashv \Delta'', r : C[\langle S_l \rangle_{l\in E}] \quad \Delta'' = \mathsf{join}(\{\Delta_l\}_{l\in E})}{\Delta \vdash \mathtt{switch}\ (e@r)\ \{e_l\}_{l\in E} : \mathsf{join}(\{U_l\}_{l\in E}) \dashv \Delta'}$$

$$\textsc{AtR}\ \frac{\Delta \vdash e : U \dashv \Delta'}{\Delta \vdash e@r : U \dashv \Delta'} \qquad \textsc{Object}\ \frac{S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \quad S \xrightarrow{\vec{s}} S'}{\Delta \vdash C[\widetilde{f : o}] : C[S'] \dashv \Delta}$$

$$\textsc{Heap}\ \frac{\forall r : U \in \Delta.\ h(r) = o \qquad \Delta \vdash o : U \dashv \Delta}{\Delta \vdash h} \qquad \textsc{Config}\ \frac{\Delta \vdash h \qquad \Delta \vdash e : U \dashv \Delta'}{\Delta \vdash h, e : U \dashv \Delta'}$$

Fig. 8. Typestate inference rules for runtime syntax.

The parameter of a method must be consumed by the body of the method. This is shown by the typing $x : \mathsf{initT}(T')$ in rule Call, where $\mathsf{initT}(T')$ is `end` if T' has a typestate. This can be done in several ways: by following the typestate of the parameter to `end`; by returning the parameter as the result of the method (perhaps after following part of its typestate); by assigning the parameter to a field (again, perhaps after following part of its typestate). Assigning a parameter to a field is always possible as a default way of consuming the typestate of a parameter.

Rule Call requires that the types of the receiver `c` in the input and output typing contexts for the body of the method are equivalent, according to the relation $=_{\mathsf{sbt}}$. This avoids exposing to the caller how the method uses its receiver. For example, assume method `pushVal` is defined as:

```
Stack pushVal(Stack x){ x.push(2); x = this.popAll(x); x }
```

Inferring a typestate for the body of `pushVal` with input context Δ_1 yields an output typing context, Δ', such that:

$$\Delta'(\mathtt{c}) = \mathtt{StackUser[Stack\ popAll(Stack) : \{\{Stack\ popAll(Stack) : end\}\}]}$$

Given that $\Delta_1(\mathtt{c}) = \mathtt{StackUser[\{\{Stack\ popAll(Stack) : end\}\}]}$, it is revealed that the body of `pushVal` calls method `popAll` on its receiver, exposing this implementation detail to the caller.

Methods, fields, classes and programs. The rules for methods, fields, classes and programs are given in Fig. 7. Rule Method-St uses the relation $\mathsf{refines}(U, T)$, which means that U adds a typestate to T if T is a class. It is defined by:

$$\begin{array}{ll} \mathsf{refines}(C[S], C) & \text{if } S \text{ is a state in } \mathsf{typestate}(C) \\ \mathsf{refines}(T, T) & \text{if } T \text{ is not a class} \end{array}$$

Rule Method-St infers a method-prefixed typestate: it first computes the continuation typestate, and then uses the output typing context to infer the method prefix, by first inferring a typestate for its body. The auxiliary definition $\mathsf{refines}(U_i, T_i)$ is used to check that the return type and parameter types declared in the syntax of the method match the corresponding inferred types for return and parameters, respectively; essentially, we expect to infer a typestate of type $C[S]$, for some S, for (return and parameter) types that are declared with type C. As in Call, a self-call should preserve the typestate of the receiver up to $=_{\mathsf{sbt}}$. Rule Enum-St is similar to rule Method-St, inferring and then joining the typestates of all branches and then inferring the method prefix. Rule Set-St requires the inference and join of the typestates of all branches. Rule End-St requires all fields of the class to finish in the `end` typestate. Rules Rec-St and Var-St are similar to rules LExpr and Continue, binding and using a recursive variable, respectively. Rule Class initiates the inference of the typestate of the class. It states that a class declaration is well-typed if every field of the class has an `end` typestate in the typing context computed in the premise of Class. Rule Program states that a program is well-typed if all of its classes are well-typed.

The inference rules for runtime expressions are given in Fig. 8. We show only those that differ from the ones in Fig. 6. Rule Switch-AtR is similar to Switch, the difference being the condition of the switch, which is evaluated to an active receiver rather than a method call. Rule AtR infers a typestate for $e@r$, by first inferring a typestate for e. The other rules are used to type runtime configurations. Rule Object, similarly to rule New, checks that the typestate of the objects in the context matches the declared typestate of their class. Rule Heap uses rule Object to check whether a typing context is consistent with all the objects in the heap. Finally, rule Config infers a typestate for a runtime configuration, by first inferring a typestate for the expression and then using its output typing context to type the heap. The output typing context and the typestate of the configuration match those of the expression.

6.2. Properties of the typestate inference system

Progress and type preservation require that the output typing context of an expression mimic the reductions of the expression itself. To this end, we define a labelled reduction relation on the typing context in Fig. 9 which use the same

$$\textsc{Ty-Id}\ \frac{}{\Delta \xrightarrow{\tau} \Delta} \qquad \textsc{Ty-If}\ \frac{\Delta' \preccurlyeq_{\mathsf{sbt}} \Delta}{\Delta \xrightarrow{\texttt{if}} \Delta'} \qquad \textsc{Ty-AsgnC}\ \frac{}{\Delta \xrightarrow{r.f=c} \Delta}$$

$$\textsc{Ty-Call}\ \frac{}{\Delta, r : C[\{T\ m(T') : S\}] \xrightarrow{r.T\ m\ T'} \Delta, r : C[S]}$$

$$\textsc{Ty-AsgnR}\ \frac{}{\Delta, r.f : C[\texttt{end}], r' : C[S] \xrightarrow{r.f=r'} \Delta, r.f : C[S], r' : C[\texttt{end}]}$$

$$\textsc{Ty-New}\ \frac{(S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \wedge \forall r.f.f' : C'[S'] \in \Delta.\ S' = \texttt{end})}{\Delta, r.f : C[\texttt{end}] \xrightarrow{r.f.\texttt{new}\ C} \Delta, r.f : C[S]}$$

$$\textsc{Ty-Label}\ \frac{\Delta' \preccurlyeq_{\mathsf{sbt}} \Delta}{\Delta, r : C[\langle S_l \rangle_{l \in E}] \xrightarrow{r.\langle l' \rangle} \Delta', r : C[S_{l'}]}$$

Fig. 9. Reduction relation on typing contexts.

labels as the reductions on expressions. Rule Ty-Id states that Δ remains unchanged under a τ-reduction. Rule Ty-New states that a path in Δ mapped to an `end` typestate reduces under $r.f.\texttt{new}\ C$ and its typestate is updated accordingly. Rules Ty-AsgnR and Ty-AsgnC label the reduction with an assignment of a path and a constant, respectively. The former ensures linearity is respected when an assignment takes place; the latter leaves the typing context unchanged. Rule Ty-Call performs a reduction of a method-prefixed typestate with the method prefix as the label. Similarly, rule Ty-Label reduces with an enumerated value for paths that have a runtime switch typestate. The behaviour of the `if` label is captured by rule Ty-If. In both the last two rules the result of the reduction is a subtype of the starting typing context.

We now state the progress and a type preservation theorem. The proof is given in Appendix A.

Theorem 4. (Progress and Type Preservation) *Assuming a program context $\widetilde{D}$, let e be a run time expression and suppose $\Delta \vdash h, e : U \dashv \Delta''$. Then, either e is a value, or there exist unique ℓ, h' and e' such that $h, e \xrightarrow{\ell} h', e'$, and there exist Δ' and U' such that $\Delta \xrightarrow{\ell} \Delta'$ and $\Delta' \vdash h', e' : U' \dashv \Delta''$ and $U' \preccurlyeq_{\mathsf{sbt}} U$.*

Type preservation implies the runtime safety property that for every object, the sequence of method calls and enumerated return values is a path within the declared typestate of the object's class. The reasoning is the following. The type preservation theorem, in the case when l is a method call or a return of an enumerated value, states that l is the first step of a path in the inferred typestate. By repeatedly using type preservation along a sequence of steps, we find that a sequence of method calls and returns is always a path within the inferred typestate. When an object is created (rule New), its inferred typestate is a subtype of its declared typestate. Subtyping is essentially simulation, so it implies trace inclusion. Therefore the sequence of method calls and enumerated return values is a path within the declared typestate.

Typability also guarantees that the program completely uses every object according to its typestate, by following a path to `end`. This does not mean that every object will be completely used at runtime, for example because there could be non-terminating code before a particular method in the typestate is called. But there must be code that can potentially follow every typestate to termination.

6.3. Typechecking expressions and typechecking programs

The proof of type preservation, and hence the runtime safety property, does not require the program $\widetilde{D}$ to be typable. That is, the rules in Fig. 7 are not used. The type preservation theorem can be applied to an initial configuration of the form described at the end of § 5, and typability of the configuration uses the rules in Figs. 6 and 8. What, then, is the purpose of the rules in Fig. 7? These rules check that the declared typestate of each class is consistent, in the sense that the sequences of method calls allowed by the typestate are consistent with the effects of the method calls on the typestates of the fields of the class. If we consider our formal system without Fig. 7, typestates are inferred and then compared with the declared typestates, but the declared typestates might be inconsistent: they might allow additional sequences of method calls that are not consistent and could never be inferred. This is not a problem for type safety, but the presence of "junk" in a declared typestate would be undesirable from a documentation point of view.

If Fig. 7 is used to check consistency of the declared typestates, then an additional property is guaranteed, which we state informally. For every class C, it is possible to construct a typable expression that creates an object of class C and uses all of the behaviour allowed by the declared typestate of C. StMungo does this when it generates the skeleton main method that uses a generated API.

Mungo implements the rules from Fig. 7 for two reasons: (1) to avoid the declaration of inconsistent and therefore confusing typestates; (2) to avoid checking method definitions at every call site, as explained in §6.1.

6.4. Implementation of the type system

The type system in this section captures most of the principles that are implemented in Mungo. However, for the sake of simplicity not all features of the implementation have been formalised.

Firstly, the formalisation omits some Java constructs that are not central to the treatment of typestate.

Secondly, the formalisation associates a typestate specification with every class, and therefore all objects are subject to linear typing in order to ensure unique references. The implementation also supports classes that do not have typestate specifications, and objects of such classes are not controlled by linear typing.

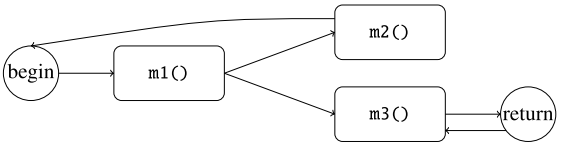
Thirdly, the most important feature not reflected in the formalisation is support for recursive method calls and a related technique for avoiding repeated analysis of method bodies. Mungo treats a method body in isolation by inferring a typestate for all objects used within the method body. It also keeps track of the objects referenced by the fields used within a method when the method is called and when the method returns. At a method call, instead of re-analysing the method body, Mungo combines the inferred typestates from the previous analysis of the method body with the current usage of the fields. This technique also supports recursive method calls, by constructing cycles. The data structure that represents a typestate resembles a state machine.

Formalising this aspect of Mungo would require defining a notion of *partial* typestate. This is because the analysis of a method leaves the usage of some objects referenced by fields incomplete, i.e. truncated. We leave formalising partial typestate to future work, but here we illustrate the operation of Mungo on a simple example.

Consider the Java code

```
public void m() {
  f.m1();
  if(cond) {
    f.m2();
    m();
  }
  f.m3();
}
```

where the body of method `void m()` uses field `f` and also makes a recursive call within a conditional statement. The Mungo typechecker constructs the following data structure to represent the typestate of field `f`.



For each method call on field `f`, there is a node labelled by the name of the method. Directed edges (arrows) between nodes represent the control flow. The initial arrow comes from a control point that designates the beginning of the method body. The continuation from `m2()` is the recursive call, which is represented by an arrow to the initial control point. The control point that designates the return of the method body has an arrow towards method `m3()`. This is because after a recursive call the control flow proceeds to the call of `m3()`. If there were no recursive call then the continuation from `f.m2()` would be `f.m3()`. A similar control flow structure is created for every field that is used inside a method. These control flow structures are what we mean by partial typestates, as the circular nodes are points at which further typestate transitions can be added when other methods are typechecked. The partial typestates can be regarded as truncated or fragmentary state machines.

7. Related and future work

Session types and programming languages. The Session Java (SJ) language [29] builds on earlier work [15,14,17] to add binary session type channels to Java. SJ has been applied to a range of situations including scientific computation [39] and event-driven programming [27]. SJ implements a library for binary sessions that have a pre-defined interface. The Java syntax is extended with communication statements that enable typechecking. The scope of a session is restricted to the body of a single method. Mungo lifts these restrictions by allowing the abstraction of multiparty session types as user-defined objects that can be passed and used throughout different program scopes. Gay et al. [23] outlined an implementation of their type system as a language called Bica, which is not currently maintained and is unusable. Mungo improves on Bica by using type inference to remove the need for typestate declarations on methods. Hu et al. [27] extend Session Java with runtime type inspection and asynchronous communication semantics to enable an event-driven framework based on binary session types. As a usecase they implement a binary session-typed SMTP server that uses a reactive structure to handle multiple clients concurrently. In our work we implement an SMTP client by using StMungo, which automatically generates code

from a global protocol. Extending Mungo with runtime typestate inspection would enable us to investigate event-driven programming with *multiparty* session types.

In Capecchi et al. [9] a class defines sessions *instead of* methods. A session generalises a method to an extended session typed dialogue over a communication channel As far as we know, this new paradigm has not yet been implemented.

Ng et al. [38] typecheck the operations of a library that implements multiparty session types using a restricted set of MPI [32] primitives. In contrast, our framework typechecks Java statements and expressions, instead of higher-level operations. Ng et al. [37] use Scribble to automatically generate MPI code based on user-defined kernels that produce and consume data. The generated code does not require typechecking. On the other hand, the StMungo transpiler can be used together with the Mungo typechecker to develop more flexible multiparty session type implementations.

Monitoring based on Scribble. Neykova et al. [36] use Scribble protocol definitions to achieve dynamic monitoring in Python, by translating local protocols into finite state machines that intercept communication and check the validity of runtime messages. Subsequently, [35] implements a session-based Actor framework that uses runtime monitoring to integrate multiparty session types. A hybrid approach is used by Hu [28] to analyse an SMTP client in Java. Hu's SMTP API implements multiparty session types using a pattern in which each communication method returns the receiver object with a new type that determines which communication methods are available at the next step. If the pattern is used properly, then standard Java typechecking can verify correctness of communication, but runtime monitoring is needed to check linearity constraints. In contrast, our analysis of SMTP is able to statically check all aspects of the protocol implementation.

The receiver-returning pattern is at the basis of functional programming with session types [22] and has been used to achieve protocol checking in Idris [30] and as a replacement for explicit typestate in Rust [42].

Typestate. There have been several projects to add typestate to practical languages, since their introduction in [45]. Vault [12,19] is an extension of C, and Fugue [13] applies similar ideas to C#. Plural [6] is based on Java and has been used to study access control systems [5] and transactional memory [4], and to evaluate the effectiveness of typestate in Java APIs [6]. In contrast Mungo follows Gay et al. which is inspired by session types; the possible sequences of method calls are explicitly defined, rather than being consequences of pre- and post-conditions. Like Plural, a typestate in Mungo can depend on the return value of a method call.

Sing# [18] is an extension of C# which was used to implement Singularity, an operating system based on message-passing. It incorporates typestate-like contracts, which are a form of session type, to specify protocols. Bono et al. [8] have formalised a core calculus based on *Sing#* and proved type safety.

Aldrich et al. [2,46] propose a new paradigm of *typestate-oriented programming*, implemented in the Plaid language. Instead of class definitions, a program consists of state definitions containing methods that cause transitions to other states. Transitions are specified in a similar way to Plural's pre- and post-conditions. Like classes, states are organised into an inheritance hierarchy. Recent work [20,48] uses gradual typing to integrate static and dynamic typestate checking. We focus on the object-oriented paradigm in order to be able to apply our results to Java.

Bodden and Hendren [7] developed the Clara framework, which combines static typestate analysis with runtime monitoring. The monitoring is based on the tracematches approach [3], using regular expressions to define allowed sequences of method calls. The static analysis attempts to remove the need for runtime monitoring, but if this is not possible, the runtime monitor is optimised. Mungo uses a purely static analysis, and can allow the state after a method call to depend on the method's (enumerated type) result.

Typestate systems must control aliasing, otherwise method calls via aliases can cause inconsistent state changes. Literature includes the "adoption and focus" approach of Vault and Fugue, the permission-based approaches of Plural and Plaid, and an expressive fine-grained system by Militão et al. [33]. Also relevant is recent work by Crafa and Padovani [11] which applies the chemical approach to concurrent typestate oriented programming, allowing objects to be accessed and modified concurrently by several processes, each potentially changing only part of their state. We expect that many of these systems can be applied to Mungo. However, linear typing has not been a limiting factor for the applications described in the present paper.

Future work. The combination of Mungo and StMungo is effective for statically checking the correct implementation of communication protocols. We intend to extend Mungo to increase its power for general-purpose programming with typestate. Our first aim is to generalise the use of linear typing as a mechanism for the alias control required by typestate systems. Candidates include the "adoption and focus" technique of Vault and Fugue, the permission-based approaches of Plural and Plaid, and the system by Militão et al. [33]. Another aim is to support generics and inheritance. Inheritance between typestate classes requires a subtyping relation between their typestate specifications, based on standard definitions of subtyping for session types [21]. Method calls on an object whose type is a generic parameter must be typechecked against the typestate specification of the parameter's upper bound. To extend typechecking to exception handlers, we need to allow typestate specifications to define the state transitions corresponding to exceptions, and check that these transitions are consistent with the states of fields at the point where an exception is thrown. Existing work on exceptions in session types [10] provides inspiration, but doesn't address the complexities of Java's exception mechanism. Using these Mungo extensions with StMungo for more sophisticated protocol verification will also require extensions to Scribble to support generic protocols, inheritance between protocols, and more general handling of exceptions.

8. Conclusion

We have presented two tools, Mungo and StMungo, which extend the Java development process with support for static typechecking of communication protocols. Mungo extends Java with typestate definitions, which associate classes with state machines defining permitted sequences of method calls. StMungo uses the typestate feature to connect Java to Scribble, the latter being a language used to specify communication protocols. In order to illustrate the practicality and robustness of Mungo and StMungo, we have implemented a substantial case study, an SMTP client, which we are able to statically typecheck. We use this client to communicate with the `gmail` server. Finally, we have formalised the essential features of Mungo by defining a typestate inference system for a core object-oriented language. We proved safety and progress properties (Theorem 4), which mean that typestate inference guarantees correct behaviour of a program with respect to the declared typestate specifications.

Acknowledgements

This research was funded by the project *From Data Types to Session Types: A Basis for Concurrency and Distribution* (UK EPSRC, grant EP/K034413/1). During the research, Dimitrios Kouzapas was employed by the University of Glasgow. We thank Laura Voinea for contributing to Mungo and StMungo. We thank Garrett Morris, Raymond Hu and Nobuko Yoshida for useful comments and discussion.

Appendix A. Progress and subject reduction

We start this section with some auxiliary results. We use $\Delta\{v : U\}$ to denote Δ where the value v is updated to the type U.

Lemma 5 *(Typability of Heap Update). Let h be a heap and r a runtime path such that $\Delta \vdash h$ and $r : U \in \Delta$.*
1. If $U \neq C[S]$ and $\Delta \vdash o : U \dashv \Delta$, then $\Delta \vdash h\{r \mapsto o\}$.
2. If $U = C[S]$ and $\Delta' \vdash o : C[S'] \dashv \Delta'$, then $\Delta' \vdash h\{r \mapsto o\}$, for $\Delta' = \Delta\{r : C[S']\}$.

Proof. Both cases follow by using rule Heap and for 1. typing rules for constants are used, and for 2. rule Object is used. □

Lemma 6 *(Replacement). If*
- *d is a derivation for $\Delta \vdash \mathcal{E}[e] : U \dashv \Delta''$,*
- *d' is a subderivation of d concluding $\Delta \vdash e : U_e \dashv \Delta_e$,*
- *the position of d' in d corresponds to the position of the hole in $\mathcal{E}$,*
- *$\Delta' \vdash e' : U_{e'} \dashv \Delta_e$, such that $U_{e'} \preccurlyeq_{\mathsf{sbt}} U_e$,*

then $\Delta' \vdash \mathcal{E}[e'] : U' \dashv \Delta''$ such that $U' \preccurlyeq_{\mathsf{sbt}} U$.

Proof. Follows [23], by replacing the derivation d' in d with the derivation for $\Delta' \vdash e' : U_{e'} \dashv \Delta_e$. □

Lemma 7 *(Substitution).*
1. If $\Delta, x : U' \vdash e : U \dashv \Delta'$ and $\Delta\{v : U'\} \vdash v : U' \dashv \Delta''$, then $\Delta\{v : U'\} \vdash e\{v/x\} : U \dashv \Delta'$.
2. Assume $\Delta_1 \vdash e : U \dashv \Delta', \lambda : X$ and $\Delta_2 \vdash e' : U \dashv \Delta'$. Then, $\Delta \vdash e\{e'/\texttt{continue}\,\lambda\} : U \dashv \Delta'$ with

$$\begin{aligned}\Delta \;=\; & \{r : C[S\{S'/X\}] \mid r : C[S] \in \Delta_1 \text{ and } r : C[S'] \in \Delta_2\} \\ \cup\; & \{r : U' \mid r : U' \in (\Delta_1 \cup \Delta_2) U' \neq C'[S']\} \cup \Delta_1 \backslash \Delta_2 \cup \Delta_2 \backslash \Delta_1\end{aligned}$$

Proof. By induction on the derivation of the typing judgement, with case analysis on the last rule. Lemma 6 is used in 2. to replace `continue` λ with e'. □

Lemma 8 *(Subtyping and join). The following relate subtyping and join on inferred types U and typing contexts Δ.*
1. Let U, U' be inferred types such that $\mathsf{join}(U, U')$ is defined. Then, $U \preccurlyeq_{\mathsf{sbt}} \mathsf{join}(U, U')$ and $U' \preccurlyeq_{\mathsf{sbt}} \mathsf{join}(U, U')$.
2. Let Δ, Δ' be such that $\mathsf{join}(\Delta, \Delta')$ is defined. Then, $\Delta \preccurlyeq_{\mathsf{sbt}} \mathsf{join}(\Delta, \Delta')$ and $\Delta' \preccurlyeq_{\mathsf{sbt}} \mathsf{join}(\Delta, \Delta')$.

Proof. The proof follows immediately by combining the definition of subtyping in Fig. 4 and the definition of join Fig. 5. □

Lemma 9 *(Typability of Subterms). If d is a derivation for $\Delta \vdash \mathcal{E}[e] : U \dashv \Delta''$ then there exist Δ' and U' such that d has a subderivation d' concluding $\Delta \vdash e : U' \dashv \Delta'$ and the position of d' in d corresponds to the position of the hole in $\mathcal{E}$.*

Proof. The proof proceeds by induction on the structure of context $\mathcal{E}$. We give the most interesting cases.

• $\mathcal{E} = r.m(\mathcal{E}')$: by assumption $\Delta \vdash r.m(\mathcal{E}'[e]) : U \dashv \Delta''$. By inversion on rule Call $\Delta \vdash \mathcal{E}'[e] : U' \dashv \Delta''', r : C[\{T\ m(T') : S\}]$, where the typing context Δ''' and the type of r are inferred by the premise of Call. We conclude by induction hypothesis on $\mathcal{E}'$.

• $\mathcal{E} = r.f = \mathcal{E}'$: by assumption $\Delta \vdash r.f = \mathcal{E}'[e] : U \dashv \Delta''$. There are two rules that can be applied for assignment, rule AsgnC and rule AsgnR. By inversion on the former we obtain $\Delta \vdash \mathcal{E}'[e] : U \dashv \Delta''', r : U$; by inversion on the latter we obtain $\Delta \vdash \mathcal{E}'[e] : C[S] \dashv \Delta''', r : C[\mathtt{end}]$, where the typing context Δ''' and the type for r are inferred by the premise of the rule. We conclude by induction hypothesis on $\mathcal{E}'$. □

Proof of Theorem 4. Assuming a program context $\widetilde{D}$, let e be a run time expression and suppose $\Delta \vdash h, e : U \dashv \Delta''$. Then, either e is a value, or there exist unique ℓ, h' and e' such that $h, e \xrightarrow{\ell} h', e'$, and there exist Δ' and U' such that $\Delta \xrightarrow{\ell} \Delta'$ and $\Delta' \vdash h', e' : U' \dashv \Delta''$ and $U' \preccurlyeq_{\mathsf{sbt}} U$. □

Proof. The proof is by induction on the structure of the expression e wrt contexts. Note that the uniqueness of ℓ, h' and e' implies that the reduction is deterministic.

We present first the inductive case. Let $e = \mathcal{E}[e_1]$ where e_1 is not a value and $\mathcal{E} \neq []$. By assumption and inversion on rule Config we have $\Delta \vdash \mathcal{E}[e_1] : U \dashv \Delta''$. By Lemma 9 there exist Δ_1 and U_1 such that $\Delta \vdash e_1 : U_1 \dashv \Delta_1$. By induction hypothesis there exist h', ℓ such that $h, e_1 \xrightarrow{\ell} h', e_2$. By induction hypothesis we also have $\Delta \xrightarrow{\ell} \Delta'$ and $\Delta' \vdash h, e_2 : U_2 \dashv \Delta_1$, which by inversion on Config means that $\Delta' \vdash h$ and $\Delta' \vdash e_2 : U_2 \dashv \Delta_1$, where $U_2 \preccurlyeq_{\mathsf{sbt}} U_1$. By rule R-Ctx we have $h, \mathcal{E}[e_1] \xrightarrow{\ell} h', \mathcal{E}[e_2]$. By Lemma 6 we obtain $\Delta' \vdash \mathcal{E}[e_2] : U' \dashv \Delta''$ with $U' \preccurlyeq_{\mathsf{sbt}} U$. We conclude by rule Config.

The base cases when e is of the form $\mathcal{E}[v]$ with $\mathcal{E}$ elementary, not being of the form $\mathcal{E}[\mathcal{E}']$ with $\mathcal{E}' \neq []$, and not of the form $\mathcal{E}[e_1]$ are in the following. If e is a value, then there is nothing to prove. If e is not a value, we give some of the most interesting cases for e wrt contexts:

- Case e is $(r.f = \mathtt{new}\ C)$. By hypothesis and typing rule R-New we have

  $$h, r.f = \mathtt{new}\ C \xrightarrow{r.f\,\mathtt{new}\ C} h\{r.f \mapsto C[\widetilde{f : \mathsf{init}(T)}]\}, *$$

  such that $\mathsf{fields}(C) = \widetilde{T\ f}$. By hypothesis and typing rule Config $\Delta \vdash h$ and $\Delta \vdash r.f = \mathtt{new}\ C : U \dashv \Delta''$. By inversion and typing rule New we have:

  $$\frac{S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \qquad \forall r.f.f' : C'[S'] \in \Delta_1 \implies S' = \mathtt{end}}{\Delta_1, r.f : C[\mathtt{end}] \vdash r.f = \mathtt{new}\ C : \mathtt{void} \dashv \Delta_1, r.f : C[S]}\ \textsc{New}$$

  where $\Delta = \Delta_1, r.f : C[\mathtt{end}]$, $U = \mathtt{void}$ and $\Delta'' = \Delta_1, r.f : C[S]$. By rule Ty-New we have

  $$\Delta_1, r.f : C[\mathtt{end}] \xrightarrow{r.f\,\mathtt{new}\ C} \Delta_1, r.f : C[S] = \Delta''$$

  such that $S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C)$ and for all fields $r.f.f' : C'[S'] \in \Delta_1$ and state $S' = \mathtt{end}$. By applying typing rule Void we have:

  $$\Delta'' \vdash * : \mathtt{void} \dashv \Delta''$$

  It remains to prove $\Delta'' \vdash h'$ namely,

  $$\Delta_1, r.f : C[S] \vdash h\{r.f \mapsto C[\widetilde{f : \mathsf{init}(T)}]\}$$

  By hypothesis $\Delta_1, r.f : C[\mathtt{end}] \vdash h$. Now we want to type the updated reference $r.f$ to $C[\widetilde{f : \mathsf{init}(T)}]$. By rule Object and an empty set of labels $\vec{s}$

  $$\frac{S \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C)}{\Delta_1, r.f : C[S] \vdash C[\widetilde{f : \mathsf{init}(T)}] : C[S] \dashv \Delta_1, r.f : C[S]}$$

  We conclude by Lemma 5.
- Case e is $r.f = r'$. By hypothesis and by rule R-AsgnR we have

  $$h, r.f = r' \xrightarrow{r.f = r'} h'\{r.f \mapsto h(r')\}, *$$

  where $h' = h\{r' \mapsto \mathtt{null}\}$. By hypothesis and by rule Config $\Delta \vdash h$ and $\Delta \vdash r.f = r' : U \dashv \Delta''$. By inversion and typing rule AsgnR

  $$\frac{\Delta \vdash r' : C[S] \dashv \Delta_1, r.f : C[\mathtt{end}]}{\Delta \vdash r.f = r' : \mathtt{void} \dashv \Delta_1, r.f : C[S]}$$

where $U = \texttt{void}$ and $\Delta'' = \Delta_1, r.f : C[S]$, and for readability we let $\Delta_2 = \Delta_1, r.f : C[\texttt{end}]$. Let $r' \neq r.f$. Since r' is a path typed by $C[S]$, the premise of the above derivation is obtained by PathR. This implies that contexts Δ and Δ_2 differ only in the typing of r'. By inversion, $\Delta(r.f) = \Delta_2(r.f) = C[\texttt{end}]$ and $\Delta(r') = C[S]$ and $\Delta_2(r') = \Delta_1(r') = C[\texttt{end}]$. By rule Ty-AsgnR

$$\Delta_3, r' : C[S], r.f : C[\texttt{end}] \xrightarrow{r.f = r'} \Delta_3, r.f : C[S], r' : C[\texttt{end}]$$

where $\Delta = \Delta_3, r' : C[S], r.f : C[\texttt{end}]$ and $\Delta' = \Delta_3, r.f : C[S], r' : C[\texttt{end}]$. Since $\Delta'' = \Delta'$, by applying rule Void we conclude $\Delta' \vdash * : \texttt{void} \dashv \Delta''$. It remains to prove that

$$\Delta_3, r.f : C[S], r' : C[\texttt{end}] \vdash h'\{r.f \mapsto h(r')\}$$

where $h' = h\{r' \mapsto \texttt{null}\}$. Recall that

$$\Delta_3, r' : C[S], r.f : C[\texttt{end}] \vdash h$$

The result follows by applying Lemma 5 for $r.f$ and r'. We conclude by Config. Let $r' = r.f$. By rewriting AsgnR with $r.f$ instead of r' we notice that the derivation holds if $S = \texttt{end}$. Then the proof proceeds trivially.

- Case e is $r.m(v)$. By hypothesis and by rule R-Call

$$h, r.m(v) \xrightarrow{r.T\ m\ T'} h, e\{v/x\}\{r/\texttt{this}\}@r$$

such that $h(r) = C[\widetilde{f : o}]$ and $T\ m(T'\ x)\ \{e\} \in \mathsf{methods}(C)$. By hypothesis and by rule Config $\Delta \vdash h$ and $\Delta \vdash r.m(v) : U \dashv \Delta''$. By inversion and typing rule Call

$$\frac{\begin{array}{c} T\ m(T'\ x)\ \{e\} \in \mathsf{methods}(C) \quad S' =_{\mathsf{sbt}} S \\ \Delta_2, r : C[S'], x : U' \vdash e\{r/\texttt{this}\} : U \dashv \Delta_1, r : C[S] \\ \Delta \vdash v : U' \dashv \Delta_2, r : C[\{T\ m(T') : S\}] \end{array}}{\Delta \vdash r.m(v) : U \dashv \Delta_1, r : C[S]}$$

where $\Delta'' = \Delta_1, r : C[S]$ and for readability let $\Delta''' = \Delta_2, r : C[\{T\ m(T') : S\}]$. Notice that $v \neq r$, otherwise the method call $r.m(v)$ would not be well-typed. Then, $\Delta(r) = \Delta'''(r) = C[\{T\ m(T') : S\}]$. Let $\Delta = \Delta_3, r : C[\{T\ m(T') : S\}]$ By rule Ty-Call we have

$$\Delta_3, r : C[\{T\ m(T') : S\}] \xrightarrow{r.T\ m\ T'} \Delta_3, r : C[S]$$

We need to prove that $\Delta_3, r : C[S] \vdash h$ and

$$\Delta_3, r : C[S] \vdash e\{v/x\}\{r/\texttt{this}\}@r : U \dashv \Delta''$$

By AtR it suffices to show $\Delta_3, r : C[S] \vdash e\{v/x\}\{r/\texttt{this}\} : U \dashv \Delta''$. By the premise of Call,

$$\Delta_2, r : C[S'], x : U' \vdash e\{r/\texttt{this}\} : U \dashv \Delta_1, r : C[S]$$
$$\Delta_3, r : C[\{T\ m(T') : S\}] \vdash v : U' \dashv \Delta_2, r : C[\{T\ m(T') : S\}]$$

it is the case that either $\Delta_2 = \Delta_3$ with $\Delta_2(v) = \Delta_3(v) = U'$, or $\Delta_2(v) = U''$ and $\Delta_3 = \Delta_2\{v : U'/v : U''\}$. By Lemma 7 we have

$$\Delta_3, r : C[S'] \vdash e\{v/x\}\{r/\texttt{this}\} : U \dashv \Delta_1, r : C[S]$$

Since $S =_{\mathsf{sbt}} S'$, we conclude by rule Equiv. By rule Heap we have

$$\frac{h(r) = C[\widetilde{f : o}] \qquad \Delta \vdash C[\widetilde{f : o}] : C[\{T\ m(T') : S\}] \dashv \Delta}{\Delta_3, r : C[\{T\ m(T') : S\}] \vdash h}$$

and by inversion of rule Object on the right-hand side premise we have

$$\frac{S_C \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \qquad \exists \vec{s}.\ S_C \xrightarrow{\vec{s}} \{T\ m(T') : S\}}{\Delta \vdash C[\widetilde{f : o}] : C[\{T\ m(T') : S\}] \dashv \Delta}$$

We perform another reduction with label $T\ m(T)$ and we obtain

$$\frac{S_C \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C) \qquad \exists \vec{s}.\ S_C \xrightarrow{\vec{s}} \{T\ m(T') : S\} \xrightarrow{T\ m(T')} S}{\Delta_3, r : C[S] \vdash C[\widetilde{f : o}] : C[S] \dashv \Delta_3, r : C[S]}$$

We now can obtain $\Delta_3, r : C[S] \vdash h$ by applying rule Heap, and conclude by rule Config.

- Case e is $\texttt{switch}\,(l'@r)\ \{e_l\}_{l\in E}$. By hypothesis and by rule R-Switch

$$h, \texttt{switch}\,(l'@r)\ \{e_l\}_{l\in E} \xrightarrow{r.\langle l'\rangle} h, e_{l'}$$

for some $l' \in E$. By hypothesis and by rule Config we have $\Delta \vdash h$ and $\Delta \vdash \texttt{switch}\,(l'@r)\ \{e_l\}_{l\in E} : U \dashv \Delta''$. By inversion and rule Switch-AtR

$$\frac{\forall l \in E \quad \Delta_l, r : C[S_l] \vdash e_l : U_l \dashv \Delta'' \qquad \Delta \vdash l' : E \dashv \Delta_1, r : C[\langle l : S_l\rangle_{l\in E}] \quad \Delta_1 = \biguplus_{l\in E} \Delta_l}{\Delta \vdash \texttt{switch}\,(l'@r)\ \{e_l\}_{l\in E} : \mathsf{join}(\{U_l\}_{l\in E}) \dashv \Delta''}$$

where $U = \mathsf{join}(\{U_l\}_{l\in E})$. By inversion and rule Enum we have that $\Delta = \Delta_1, r : C[\langle l : S_l\rangle_{l\in E}]$. By Ty-Label

$$\Delta_1, r : C[\langle S_l\rangle_{l\in E}] \xrightarrow{r.\langle l'\rangle} \Delta_{l'}, r : C[S_{l'}]$$

where $l' \in E$ and $\Delta' \preccurlyeq_{\mathsf{sbt}} \Delta$. By Lemma 8 we have that $U_{l'} \preccurlyeq_{\mathsf{sbt}} \mathsf{join}(\{U_l\}_{l\in E})$. The typing judgement $\Delta_{l'}, r : C[S_{l'}] \vdash e_{l'} : U_{l'} \dashv \Delta''$ holds by the premise of Switch for $l' \in E$. We need to prove that $\Delta_{l'}, r : C[S_{l'}] \vdash h$. Recall that, by hypothesis $\Delta \vdash h$. By Heap and Object it means that there exist $\vec{s}$, such that $S_C \preccurlyeq_{\mathsf{sbt}} \mathsf{typestate}(C)$ and $S_C \xrightarrow{\vec{s}} \langle l : S_l\rangle_{l\in E}$ and

$$\Delta \vdash C[\widetilde{f : o}] : C[\langle l : S_l\rangle_{l\in E}] \dashv \Delta$$

By Definition 3, we have $\langle l : S_l\rangle_{l\in E} \xrightarrow{l'} S_{l'}$. By applying rule Object on this reduction, we have

$$\Delta_{l'}, r : C[S_{l'}] \vdash C[\widetilde{f : o}] : C[S_{l'}] \dashv \Delta_{l'}, r : C[S_{l'}]$$

We conclude by rules Heap and Config. □

References

[1] Mungo webpage, http://www.dcs.gla.ac.uk/research/mungo/.
[2] J. Aldrich, J. Sunshine, D. Saini, Z. Sparks, Typestate-oriented programming, in: Companion to the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM, 2009, pp. 1015–1022.
[3] C. Allan, P. Avgustinov, A.S. Christensen, L.J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, in: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM, 2005, pp. 345–364.
[4] N.E. Beckman, K. Bierhoff, J. Aldrich, Verifying correct usage of atomic blocks and typestate, in: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM, 2008, pp. 227–244.
[5] K. Bierhoff, J. Aldrich, Modular typestate checking of aliased objects, in: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM, 2007, pp. 301–320.
[6] K. Bierhoff, N.E. Beckman, J. Aldrich, Practical API protocol checking with access permissions, in: Proceedings of the 23rd European Conference on Object Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 5653, Springer, 2009, pp. 195–219.
[7] E. Bodden, L.J. Hendren, The Clara framework for hybrid typestate analysis, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 307–326.
[8] V. Bono, C. Messa, L. Padovani, Typing copyless message passing, in: Proceedings of the 20th European Symposium on Programming, ESOP, in: Lecture Notes in Computer Science, vol. 6602, Springer, 2011, pp. 57–76.
[9] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, Amalgamating sessions and methods in object-oriented languages with generics, Theor. Comput. Sci. 410 (2009) 142–167.
[10] M. Carbone, K. Honda, N. Yoshida, Structured interactional exceptions in session types, in: Proceedings of the 19th International Conference on Concurrency Theory, CONCUR, in: Lecture Notes in Computer Science, vol. 5201, Springer, 2008, pp. 402–417.
[11] S. Crafa, L. Padovani, The chemical approach to typestate-oriented programming, in: Proceedings of the 30th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM, 2015, pp. 917–934.
[12] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, ACM, 2001, pp. 59–69.
[13] R. DeLine, M. Fähndrich, Typestates for objects, in: Proceedings of the 18th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 3086, Springer, 2004, pp. 465–490.
[14] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, N. Yoshida, Objects and session types, Inf. Comput. 207 (5) (2009) 595–641.
[15] M. Dezani-Ciancaglini, E. Giachino, S. Drossopoulou, N. Yoshida, Bounded session types for object-oriented languages, in: Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO, in: Lecture Notes in Computer Science, vol. 4709, Springer, 2006, pp. 207–245.
[16] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, S. Drossopolou, Session types for object-oriented languages, in: Proceedings of the 20th European Conference on Object-Oriented Programming, in: Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 328–352.
[17] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, S. Drossopolou, A distributed object-oriented language with session types, in: Proceedings of the International Symposium on Trustworthy Global Computing, TGC, in: Lecture Notes in Computer Science, vol. 3705, Springer, 2005, pp. 299–318.
[18] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J.R. Larus, S. Levi, Language support for fast and reliable message-based communication in Singularity OS, in: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, ACM, 2006, pp. 177–190.
[19] M. Fähndrich, R. DeLine, Adoption and focus: practical linear types for imperative programming, in: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, ACM, 2002, pp. 13–24.
[20] R. Garcia, É. Tanter, R. Wolff, J. Aldrich, Foundations of typestate-oriented programming, ACM Trans. Program. Lang. Syst. 36 (4) (2014) 12:1–12:44.
[21] S.J. Gay, M.J. Hole, Subtyping for session types in the pi calculus, Acta Inform. 42 (2/3) (2005) 191–225.

[22] S.J. Gay, V.T. Vasconcelos, Linear type theory for asynchronous session types, J. Funct. Program. 20 (1) (2010) 19–50.
[23] S.J. Gay, V.T. Vasconcelos, A. Ravara, N. Gesbert, A.Z. Caldeira, Modular session types for distributed object-oriented programming, in: Proceedings of the 37th ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages, POPL, ACM, 2010, pp. 299–312.
[24] G. Hedin, An introductory tutorial on JastAdd attribute grammars, in: Generative and Transformational Techniques in Software Engineering III, in: Lecture Notes in Computer Science, vol. 6491, Springer, 2011, pp. 166–200.
[25] K. Honda, V. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: Proceedings of the 7th European Symposium on Programming, ESOP, in: Lecture Notes in Computer Science, vol. 1381, Springer, 1998, pp. 122–138.
[26] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: Proceedings of the 35th ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages, POPL, ACM, 2008, pp. 273–284.
[27] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, K. Honda, Type-safe eventful sessions in Java, in: Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 6183, Springer, 2010, pp. 329–353.
[28] R. Hu, N. Yoshida, Hybrid session verification through endpoint API generation, in: Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering, FASE, in: Lecture Notes in Computer Science, vol. 9633, Springer, 2016, pp. 401–418.
[29] R. Hu, N. Yoshida, K. Honda, Session-based distributed programming in Java, in: Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 5142, Springer, 2008, pp. 516–541.
[30] Idris language homepage, www.idris-lang.org.
[31] D. Kouzapas, O. Dardha, R. Perera, S.J. Gay, Typechecking protocols with Mungo and StMungo, in: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, PPDP, ACM, 2016, pp. 146–159.
[32] Message Passing Forum, MPI: A Message-Passing Interface Standard, Technical Report, University of Tennessee, 1994.
[33] F. Militão, J. Aldrich, L. Caires, Aliasing control with view-based typestate, in: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTfJP, ACM, 2010.
[34] M. Neubauer, P. Thiemann, An implementation of session types, in: Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages, PADL, in: Lecture Notes in Computer Science, vol. 3057, Springer, 2004, pp. 56–70.
[35] R. Neykova, N. Yoshida, Multiparty session actors, in: Proceedings of the 16th IFIP WG 6.1 International Conference on Coordination Models and Languages, COORDINATION, in: Lecture Notes in Computer Science, vol. 8459, Springer, 2014, pp. 131–146.
[36] R. Neykova, N. Yoshida, R. Hu, SPY: local verification of global protocols, in: Proceedings of the 4th International Conference on Runtime Verification, RV, in: Lecture Notes in Computer Science, vol. 8174, Springer, 2013, pp. 358–363.
[37] N. Ng, J.G. de Figueiredo Coutinho, N. Yoshida, Protocols by default — safe MPI code generation based on session types, in: Proceedings of the 24th International Conference on Compiler Construction, CC, in: Lecture Notes in Computer Science, vol. 9031, Springer, 2015, pp. 212–232.
[38] N. Ng, N. Yoshida, K. Honda, Multiparty session C: safe parallel programming with message optimisation, in: Proceedings of the 50th International Conference on Objects, Models, Components, Patterns, TOOLS, in: Lecture Notes in Computer Science, vol. 7304, Springer, 2012, pp. 202–218.
[39] N. Ng, N. Yoshida, O. Pernet, R. Hu, Y. Kryftis, Safe parallel programming with Session Java, in: Proceedings of the 13th International Conference on Coordination Models and Languages, COORDINATION, in: Lecture Notes in Computer Science, vol. 6721, Springer, 2011, pp. 110–126.
[40] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[41] R. Pucella, J.A. Tov, Haskell session types with (almost) no class, in: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, ACM, 2008, pp. 25–36.
[42] Rust language homepage, www.rust-lang.org.
[43] Scribble project homepage, www.scribble.org.
[44] Extended simple mail transfer protocol, RFC 5321, https://tools.ietf.org/html/rfc5321.
[45] R.E. Strom, S. Yemini, Typestate: a programming language concept for enhancing software reliability, IEEE Trans. Softw. Eng. 12 (1) (1986) 157–171.
[46] J. Sunshine, K. Naden, S. Stork, J. Aldrich, É. Tanter, First-class state change in Plaid, in: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, ACM, 2011, pp. 713–732.
[47] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: Proceedings of the 6th International Conference on Parallel Languages and Architectures Europe, PARLE, in: Lecture Notes in Computer Science, vol. 817, Springer, 1994, pp. 398–413.
[48] R. Wolff, R. Garcia, E. Tanter, J. Aldrich, Gradual typestate, in: Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 6813, Springer, 2011, pp. 459–483.
[49] N. Yoshida, R. Hu, R. Neykova, N. Ng, The Scribble protocol language, in: Proceedings of the 8th International Symposium on Trustworthy Global Computing, TGC, in: Lecture Notes in Computer Science, vol. 8358, Springer, 2013, pp. 22–41.